RMX/80
USER'S GUIDE

Manual Order Number: 9800522C

The following table provides a summary of the changes that have been incorporated within this revision. Major changes are those that caused a substantial amount of text to be modified or added; minor changes are those that had less impact on the text. The primary motivation for producing the revision was the addition of the Bootstrap Loader and the iSBC 204 Disk Controller. Sections of the document affected by these two additions are indicated in the rightmost two columns of the table.

CHANGE SECTION	DEGREE	CAUSES			
		TYPOGRAPHICALS AND CLARITY	TECHNICAL ACCURACY	BOOTSTRAP LOADER	iSBC 204 CONTROLLER
PREFACE	minor			●	
CHAPTERS					
1	minor	●	●	●	
2	minor	●	●		
3	minor	●	●		
4	minor	●	●		
5	minor	●	●		
6	minor	●	●		
7	minor	●	●		●
8	minor	●	●		
9	major			●	●
APPENDICES					
A	major	●		●	●
B	minor	●	●	●	●
C	minor	●			
D	major	●		●	●
E	minor	●	●		
F	minor	●	●		
G	minor	●			
H	minor	●			
I	minor	●			
J	minor	●	●		

RMX/80 is a user-configurable software package that provides a Real-time, Multitasking Executive for Intel Single Board Computer systems. It consists of a central Nucleus and a number of extensions that may be selected as required; together, these serve as a base for real-time application systems. RMX/80 operates on hardware systems based on the iSBC 80/20, 80/30, and 80/10 computers.

This manual is intended to serve two purposes: as an introduction to RMX/80 for the new user, often an applications programmer with little or no experience with operating systems or real-time programming; and as a reference manual supplying all the information necessary for interfacing with the RMX/80 Nucleus and extensions.

Because of the intermixing of conceptual and detailed information, at least two readings of the User's Guide are recommended. On the first reading, you should skim over the detailed information and concentrate on gaining an understanding of the kinds of services that RMX/80 provides. On subsequent readings you can study the details of how to use these services.

This User's Guide assumes that the reader is familiar with Intel Single Board Computer (iSBC) hardware, with programming techniques for 8080/8085-based systems, and with the ISIS-II conventions for program relocation and linking. (ISIS-II is the diskette operating system for the Intellec and Intellec Series II microcomputer development systems.)

Manual Organization

This manual contains the following chapters and appendices:

- "Chapter 1. Overview" which provides an introduction to real-time systems in general and RMX/80 in particular.
- "Chapter 2. RMX/80 Nucleus," which describes the RMX/80 Nucleus and its primitive operations, explains how user tasks and hardware interrupts interface with the Nucleus, and outlines the basics of Nucleus internal operation (as a reference for debugging application systems).
- "Chapter 3. Implementing an RMX/80 System," which explains how to code application tasks, provide configuration information, and combine these with RMX/80 software.
- "Chapter 4. Terminal Handler," which describes the use of the Terminal Handler extension.
- "Chapter 5. Free Space Manager," which describes the use of the Free Space Manager extension.
- "Chapter 6. Debugger," which describes the use of the Debugger extension.
- "Chapter 7. Disk File System," which describes the use of the Disk File System (DFS) extension.
- "Chapter 8. Analog Handlers," which describes the use of the Analog Handlers extension.
- "Chapter 9. Bootstrap Loader," which describes the use of the Bootstrap Loader extension.

- "Appendix A. RMX/80 Diskette Files," which lists the files on the RMX/80 product diskettes and briefly describes each.
- "Appendix B. Publics and Externals," which lists the public identifiers declared by, and the external identifiers referenced by, the RMX/80 Nucleus and extensions.
- "Appendix C. Message Types," which lists all the message types defined by RMX/80 and the numeric and PL/M symbolic values assigned to each type.
- "Appendix D. Memory Requirements," which gives the byte requirements (code and data) for all configurable parts of the RMX/80 software, including table requirements.
- "Appendix E. DFS File Structure," which describes the file structure used by the RMX/80 Disk File System, which is (in the case of standard-sized single-density or double-density diskettes) compatible with ISIS-II.
- "Appendix F. Hardware Considerations," which provides certain hardware installation instructions that must be followed when developing and operating an RMX/80 application system on Intel iSBC hardware.
- "Appendix G. iSBC 80/I0 Time Base Considerations," which gives special instructions for providing RMX/80 timing functions on systems based on the iSBC 80/10 processor board.
- "Appendix H. System Deadlock," which explains how deadlock may arise in an RMX/80-based system, and provides suggestions and further references for solution of the deadlock problem.
- "Appendix I. Demonstration System," which supplies instructions for using the Demonstration System provided with RMX/80.
- "Appendix J. System Example: A Real-Time Clock," which gives a description, flowcharts, and listings of an example RMX/80 system that includes the input-output Terminal Handler, the active Debugger, and three user tasks.

Related Publications

The use of Intel Single Board Computer hardware, programming techniques for 8080/8085-based systems, and ISIS-II relocation and linking conventions are discussed in the following manuals:

- *System 80/20-4 Microcomputer Hardware Reference Manual*, 9800484.
- *System 80/30 Microcomputer Documentation Package*, 9800707.
- *System 80/10 Microcomputer Hardware Reference Manual*, 9800316.
- *8080/8085 Assembly Language Programming Manual*, 9800301.
- *PL/M-80 Programming Manual*, 9800268.
- *ISIS-II User's Guide*, 9800306.

You will also generally need to use the ISIS-II PL/M compiler and/or the ISIS-II 8080/8085 assembler; and you may wish to use one of the Intel In-Circuit Emulators, ICE-80 or ICE-85, during system development. The following Intel manuals provide the information you need to use these products:

- *ISIS-II PL/M-80 Compiler Operator's Manual*, 9800300.
- *ISIS-II 8080/8085 Macro Assembler Operator's Manual*, 9800292.
- *In-Circuit Emulator/80 Operator's Manual*, 9800185.
- *ICE-85 In-Circuit Emulator Operating Instructions for ISIS-II Users*, 9800463.

In addition, certain RMX/80 extensions (including the Disk File System and the Analog Handlers) are designed to interface with specific Intel Single Board Computer hardware in addition to the processor boards. Reference is made in the appropriate chapters to the manuals required for the use of this additional hardware.

int_el

CONTENTS

intel

CONTENTS (Cont'd.)

intel

TABLES

int_e_l

TABLES (Cont'd.)

int_e_l

ILLUSTRATIONS

Purpose of RMX/80

RMX/80, Intel's Real-time, Multitasking EXecutive, provides a base around which you can build a real-time, multitasking application system. RMX/80 allows you to concentrate your major programming efforts on your particular application rather than on the problems of synchronizing multiple real-time tasks.

RMX/80 is designed for Intel Single Board Computer (iSBC) systems based on the iSBC 80/20, iSBC 80/30, or iSBC 80/10. The software, if made ROM-resident, is designed to initialize the system without a need for external storage media. Because RMX/80 has been designed specifically for Intel Single Board Computers, its memory requirements are appropriate for the microcomputer environment. The basic Nucleus requires less than 2K bytes of program storage and approximately 250 bytes of data storage.

RMX/80 Features

As can be seen in figure 1-1, RMX/80 is not a single, indivisible product. The heart of the system — the RMX/80 Nucleus — is supported by a number of additional software tools, or *extensions*, such as the Terminal Handler, Free Space Manager, Debugger, and Disk File System, which are designed to meet common real-time system requirements. Each RMX/80 extension is supplied as a separate module (task) or set of modules, so that you may select only those features you need for your particular application. To this collection of RMX/80 tasks you then add your application tasks.
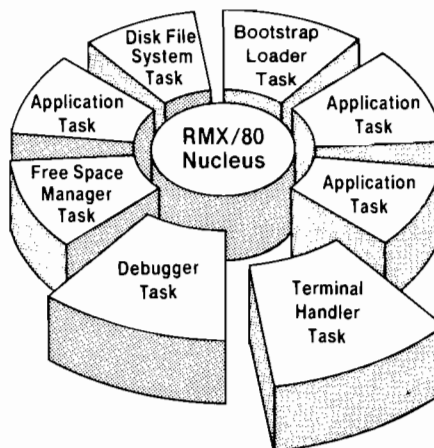
Figure 1-1. Modular Structure of a Typical RMX/80 System

The following paragraphs briefly describe the RMX/80 Nucleus and extensions. Each of these features is covered in detail in its own chapter later in this manual.

RMX/80 Nucleus

The RMX/80 Nucleus directs and coordinates system operation. It initializes system control structures, determines which task should be executed next, supplies timing functions for the system, and provides a means for controlling and exchanging data within the system. User tasks communicate with the Nucleus by means of a set of RMX/80 primitive operations.

Terminal Handler

The Terminal Handler provides real-time asynchronous I/O between an operator's terminal and tasks running under the RMX/80 Nucleus. It includes a line-edit capability similar to that of ISIS-II and an additional type-ahead feature. (ISIS-II is the diskette supervisor system used on the Intellec Development System.) The Terminal Handler may be configured with full capabilities, for output only, or in either of two minimal versions that require considerably less code space.

Free Space Manager

The Free Space Manager maintains a pool of free RAM and allocates memory from that pool upon request from a task. The Free Space Manager also allows memory to be returned to the pool when it is no longer needed.

Debugger

The Debugger is designed specifically for debugging systems running under the RMX/80 Nucleus. A special debugger is needed for such systems, since the real-time software environment changes constantly in response to external events. The Debugger allows convenient access to RMX/80 data structures. It can be run in the Single Boards Computer's memory, or an in-circuit emulator such as ICE-80 or ICE-85 can be used to load and execute the Debugger. This last option allows use of the Debugger without imposing any memory requirements on the iSBC 80.

If desired, the Debugger can be configured into an application system to provide for troubleshooting in the field. A limited-capability version of the Debugger is available for applications in which memory space is limited.

Disk File System

The Disk File System provides diskette processing capabilities which are similar to those available in ISIS-II. Files may be created, deleted and updated; data can be accessed sequentially and directly ("randomly").

Analog Handlers

The Analog Handlers provide analog input and output services for users whose systems interface with the iSBC 711, iSBC 724, and iSBC 732 analog boards. The Input Handler task performs repetitive, sequential, and random channel input; the Output Handler task performs random channel output.

Bootstrap Loader

The Bootstrap Loader allows you to store your applications software on the disk rather than in ROM. This affords you two types of flexibility. First, since your applications software is stored on disk rather than in ROM, changes are easier to implement. Modifications to your applications software entail changing the contents of the disk rather than the contents of ROM. Second, since the disk has a much larger storage capacity than does the ROM, you can store multiple applications on disk and use the Bootstrap Loader to bring them into RAM as they are needed. Consequently, you can use a single iSBC for more than one application.

The RMX/80 Product

The RMX/80 product is supplied as a set of diskette files which include all the software (except ISIS-II and Intel language translators) required to prepare an RMX/80-based application system. The following types of files are included:

- Relocatable and linkable object code for the RMX/80 Nucleus and extension tasks
- INCLUDE files containing sections of PL/M code commonly needed in programming application systems
- The PL/M library and a special RMX/80 unresolved reference library, required at link time to resolve external references
- Assembly-language macros to aid in the preparation of the configuration module for your application system
- Files to support the RMX/80 Demonstration System, a sample program furnished for demonstration and checkout of the RMX/80 software (refer to Appendix I for description and operating instructions)

Appendix A provides a list of all files on the RMX/80 diskettes, with a brief description of each.

Characteristics of Real-Time Systems

The simplest processing systems are *batch systems* . In a batch system, a number of operations are submitted at once to the processor and then performed in the sequence submitted (or according to some other scheduling scheme) without respect to external events. In contrast, a *real-time system* responds to certain events in the "real world" as they occur. Interactive terminal systems and computerized process control systems are examples of real-time systems.

Real-time processing imposes special requirements on the operating system that coordinates and schedules processing functions, as well as on the application program modules that perform these functions. This section of the chapter is provided to familiarize the user with the fundamental characteristics of real-time systems, with some specific references to the RMX/80 implementation. A basic understanding of these concepts is necessary in building an application system based on RMX/80.

Asynchronous Operation and Synchronization

Events to which a real-time system must respond may occur at unpredictable times and in unpredictable order. A system that responds to such events as they occur is said to execute *asynchronously.* Asynchronous operation requires *synchronization* of system functions with the individual *events* rather than with a system clock; i.e., one event (external to the system) causes another (the execution of a processing function) to occur. It must be noted, however, that a system operating asynchronously may still need timing functions that require a clock.

As an example of synchronization, consider a microcomputer-controlled system that weighs and stamps packages. One part of the system detects (by means of a sensor) the arrival of a package, weighs the package, and calculates pricing data. Price and weight data are then communicated to the stamping station. Since the amount of time required to move the package from the weighing station to the stamping station is known, the task that drives the stamping mechanism can wait the proper time interval and then trigger the stamping process.

In this example, the weighing and price calculation program is started when a package arrives at the weighing station; i.e., weighing and price calculation operations are synchronized with the arrival of a package. The stamping process, similarly, is synchronized with the arrival of price and weight data at the stamping station — but only after a clock-driven timed wait operation is completed.

The RMX/80 software provides for task synchronization by means of the message/exchange facility and the timed wait operation. Both of these features are described later in this chapter.

Concurrency

In a real-time system, several operations will generally be in progress at a given time. This *concurrency* of events is a distinguishing characteristic of real-time systems. As an example, consider the package weighing and stamping system discussed in the previous section. If a second package arrives at the weighing station about the same time as the first package is being stamped, the task directing the weighing of the second package may be performed concurrently with the task controlling the stamping of the first.

When there is only one central processing unit, as in a system running under RMX/80, only one operation can have control at any given instant; however, it is not necessary that that operation be finished before a faster or more important operation takes control. The speed of the processor can make concurrent operations appear to be running simultaneously.

The designer of a real-time system should identify the functions in his design that can be performed concurrently. These functions are known as *tasks* . The designer or programmer defines these tasks to a central operating system such as the RMX/80 Nucleus, which schedules and controls their operation.

While the system is running, a task with an urgent need for the system's computing resources can preempt the currently executing task. When the new task has completed its function (or must wait for another event or input), the central operating system automatically either continues the previously executing task or starts executing some other task that currently has a greater need for the processor.

For example, in a process control application the system might monitor the temperature of a paper feedstock and gauge the thickness of the stock as it passes through rollers. The system controls the process by varying the feedstock heaters and/or pressure rollers as required. Such a system would probably also display information about the process at an operator's station; the operator must be able to enter new temperature and roller pressure specifications at any time.

If this system is viewed as a series of sequential events, as it might be in a single, large program, the system performs the following functions:

- Monitor feedstock temperature
- Control heaters as required
- Update operator's display
- Monitor thickness
- Control pressure rollers as required
- Update operator's display
- Accept input from operator's console
- Control heaters and rollers as required
- Update operator's display
- Return to first step and begin again

Notice that in this "monolithic" approach, the processor cycles through the same code over and over. Considerable time can be spent checking for the occurrence of events that are asynchronous. For example, although the operator would normally enter data sporadically, the program checks for this every time it goes through its cycle. Notice also, that the program is only "ready" to accept operator data at a fixed time—it is not capable of providing flexible response to varying conditions.

Using RMX/80 in this process control application, the programmer might identify the following concurrent events:

- Monitor feedstock temperature
- Control temperature
- Monitor thickness
- Control pressure rollers
- Update operator's display
- Accept input from operator's console

The programmer would code a separate task for each concurrent event. RMX/80 provides the coordination required to execute each task only as it is needed, that is, in response to an event. In this way the system can respond to varying load conditions and throughput potential is increased.

The speed of the processor enables the system to appear to be performing multiple tasks concurrently. A microprocessor executes thousands of instructions each second. Contrast this with an operator making an entry from a console keyboard. The operator probably enters only a few characters a second. The intervals between keystrokes give the system enough time to execute several of its monitor and control tasks.

Priority

When several tasks are competing for the central processor, the real-time system must determine which task is to run first — i.e., which task has highest *priority* . The system designer or programmer must therefore assign each task a priority level for use by the system. RMX/80 provides 256 software priority levels for tasks.

Generally, tasks invoked by emergency conditions need to have higher priority than routine operations; tasks that execute rapidly or require quick response time require higher priority than slower or less urgent tasks; and tasks that service interrupts from peripheral devices should usually have higher priority than processing tasks invoked by other tasks.

In our process control application, for example, priorities might be assigned as follows:

1. Monitor thickness
2. Control pressure rollers
3. Monitor temperature
4. Control temperature
5. Accept input from operator's console
6. Update operator's display

At any given time, the running task — i.e., the task that has control of the processor — is the highest-priority task that is ready to run. The running task will continue to run until it either voluntarily surrenders control of the processor or is preempted by a newly-readied task of higher priority. When the latter situation occurs, the system

must save all relevant information about the preempted task (i.e., perform a "context save") so that the task can eventually resume processing as though it were never interrupted.

Communication

Another requirement for a real-time system is that tasks be able to communicate with each other. One task may need to invoke another and/or send it certain information. For instance, if the operator in our process control example enters a new temperature specification, the console input task must invoke the temperature control task, supplying it with the new temperature. The temperature control task, since it has a higher priority, will immediately preempt the console input task. RMX/80 provides for intertask communication by means of the message/exchange facility, which is described in the following paragraphs.

Messages and Exchanges. A *message* is a collection of data that one task communicates — i.e., "sends" — to another task. An *exchange* is a location where RMX/80 queues a list of tasks waiting for messages or a list of messages waiting to be received by tasks.

To avoid the overhead required to move an entire message to an exchange when it is sent, RMX/80 merely posts the address of the message. When a message arrives at an exchange where tasks are waiting, it is matched with the first waiting task on the list, and that task is removed from the list. Likewise, when a task arrives at an exchange where messages are waiting, it is matched with the first waiting message, and that message is removed from the list. Task-message communication at an exchange is thus first-in/first-out. When a task and a message are matched at an exchange, the task is also made ready for execution.

Note that, as would be expected from the preceding explanation, tasks and messages can never be queued at an exchange at the same time. Also note that a task cannot wait at more than one exchange at a time. This is true because of the way tasks and messages are queued at exchanges, as discussed under "Details of System Operation" in Chapter 2.

A special set of *interrupt exchanges* is reserved for use with the hardware interrupt levels (or the software pseudo-interrupt levels on the iSBC 80/10). These differ from the other exchanges in certain ways, which will be discussed more fully in Chapter 2.

Message-Consuming and Message-Producing Tasks. Most tasks can be classified as message-consuming or message-producing tasks or a combination of the two. The processing flow of these types of tasks can be shown as follows:
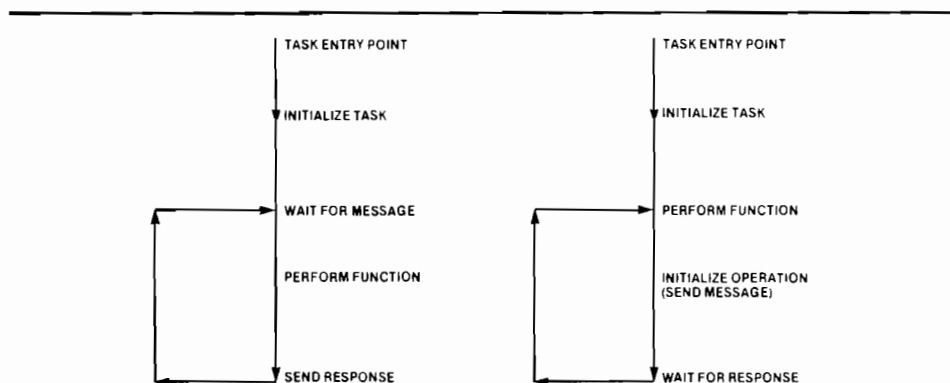
Figure 1-2. Schematic of Message-Consuming and Message-Producing Tasks

A message-consuming task waits for a message to be posted at a particular exchange and takes control of the processor only when it has received a message. The message-consuming task performs some action based upon the message and then simply resumes waiting until the next message is received. Usually, the message-consuming task acknowledges completion of its function by returning the message to an exchange where the sending task is waiting for a response.

A message-producing task initiates its function by sending a message and then surrenders control of the processor. The task continues to wait until it receives a response to its message.

Notice that the distinction between these types of tasks is not absolute, since most tasks both produce and consume messages. However, the producer-consumer concept helps clarify the general structure of tasks. Also note that because communication via an exchange implies a first-in/first-out queue of messages, buffering of the input data to a consumer task is accomplished automatically. (The message queue serves as a buffer.) Thus, a high-priority producer task can be allowed to temporarily "get ahead" of a low-priority consumer task without loss of data.

Task States. Thus far, it has been implied that tasks may be in one of two states: running or waiting. In addition to these states, a task may also be ready or suspended. For convenience, all four task states are defined here.

The *running task* is the task currently in operation.

A *ready task* is one that is ready to run. Any wait for a message or time delay has been satisfied. The running task is always the ready task with the highest current priority.

A *waiting task* is a task that is waiting at an exchange.

A *suspended task* is one that is needed within the system, but that the user wishes to prevent from competing for system resources. This occurs most commonly during testing and debugging. Suspending individual tasks simplifies the real-time environment and allows you to concentrate on the portion of the system being tested. Another possible reason for suspending a task is that it is no longer needed. For example, a task that performs initialization functions might suspend itself when it has finished its processing.

The RMX/80 software maintains a variety of control structures that keep track of all the tasks in the system. Figure 1-3 illustrates the various task states and their interaction.
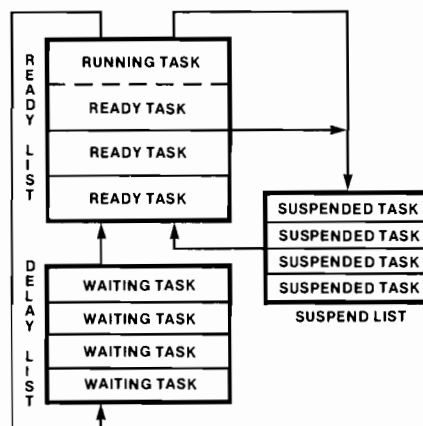
Figure 1-3. Task States

The *Ready List* is the list of all the ready tasks in the system. The *Delay List* is the list of all tasks currently in a timed wait (waits may be timed or unconditional); and the *Suspend List* is the list of all suspended tasks.

Tasks are always entered on the Ready List in order of priority. The running task is always the highest priority task on the Ready List. When several tasks have the same priority, they are entered on the Ready List in a first-in/first-out sequence. Tasks can be entered on the Ready List for any of three reasons:

- A task is ready for execution when it is first created. Tasks are created either during system initialization or by other tasks at run time.

- A task becomes ready when a condition for which it has been waiting has been satisfied. A task may wait for a message at an exchange or may wait for some specific time interval to pass or both.

- A suspended task becomes ready when it is allowed to resume competition for system resources. A task on the Suspend List that is resumed is entered directly on the Ready List.

A task is added to the Delay List when it requests a timed wait. Tasks are entered on the Delay List in order of increasing time remaining to satisfy the delay request. (The first task on the list has the least remaining time.) A task is removed from the Delay List when it "times out," or when a message arrives at the exchange at which the task is waiting, whichever occurs first.

A task is added to the Suspend List if it is ordered suspended by another task or if it suspends itself. The Suspend List is not ordered. If a waiting task is ordered suspended, RMX/80 waits for the task to become ready before moving it to the Suspend List. A task on the Suspend List that is resumed is entered directly on the Ready List.

Real-Time Programming for an RMX/80 Application

Programming for the real-time environment involves certain techniques, some of which may not be familiar to the applications programmer. This section introduces those aspects of programming which are peculiar to real-time systems as implemented with RMX/80 software. The details needed for actual programming are supplied in Chapters 2 and 3.

Perhaps the most important real-time programming technique is the dividing of an application into tasks. Users who are accustomed to modular programming will find the transition to tasking relatively natural and simple. Users who are accustomed to writing a single, large program for applications should familiarize themselves with modular programming by reading the appropriate sections of the PL/M-80 Programming Manual, 9800268, and the 8080/8085 Assembly Language Programming Manual, 9800301. Many references on the subject of modular programming are available commercially.

User tasks must be coded using an RMX/80-compatible Intel language translator. It is assumed that the programmer is familiar with the language he is using; language details are supplied in the appropriate programming, language reference, and/or ISIS-II translator manuals listed in the Preface to this manual. Examples in this manual are presented for the most part in PL/M, and in some cases also in 8080/8085 assembly language. Sufficient information is given in this manual to enable programmers to code application tasks and the configuration module in either PL/M or assembly language.

Identifying Tasks

One of the real-time programmer's first jobs is to break down his application into tasks. As described under "Concurrency" earlier in this chapter, tasks are units of processing that can be run concurrently, and which are scheduled for execution by the RMX/80 software. Careful identification of tasks is critical to the performance of an RMX/80 application system.

Identifying tasks is basically a matter of subdividing a large application into a series of smaller, logically independent programs. Frequently, input/output requirements define tasks. In our sample process control application, the temperature and roller pressure monitors and the console input and output tasks are all directly associated with input and output. The temperature and roller control tasks perform some calculations and then output the results to the appropriate device.

In general, it is better to have a number of smaller tasks rather than a few very large tasks. However, the number of tasks should not be too great, either. Too many tasks add needless complication to the system, making it difficult to modify or debug, and also add to system overhead.

Using Basic RMX/80 Nucleus Operations

The RMX/80 Nucleus provides a set of primitive operations to facilitate programming of real-time tasks. The programmer must become familiar with the use of these operations, of which the two most important — RQWAIT (Wait for Message) and RQSEND (Send Message) — are described briefly here. (Refer to Chapter 2 for complete definitions of all Nucleus operations and instructions for their use.)

The RQWAIT Operation. Issuing an RQWAIT (Wait for Message) causes the calling task to wait for a message at an exchange or (optionally) for a specified time interval to elapse. The operation requires two parameters, an exchange address and a number of system time units; the latter may be zero, indicating an indefinite wait. When the task receives a message, RQWAIT returns the address of the message. If a time limit is specified and no message becomes available at the exchange before the time limit expires, the operation returns the address of a system message that identifies the time- out condition. The timing feature enables the programmer to ensure that a task will not wait at an exchange for longer than a specified amount of time. Conversely, precise timing intervals can be obtained by performing a timed wait at an exchange to which no messages are sent.

As an example, consider an output display task that must wait for a message. Exchanges are defined by the user, and are normally addressed symbolically; for instance, we might use the name DISPEX for the exchange in our example. Let PTR be the variable into which the returned message address is to be stored. The time limit is specified as a number of system time units (50 milliseconds); a one-second wait is specified as 20 time units. Thus a call to RQWAIT specifying a wait of one second might be programmed as follows:

PL/M	8080/8085 Assembly Language	
PTR=RQWAIT(.DISPEX,20);	LXI	B,DISPEX
	LXI	D,20
	CALL	RQWAIT
	SHLD	PTR

The RQSEND Operation. RQSEND (Send Message) sends a message to an exchange by posting the address of the message at the exchange. The operation requires two parameters, an exchange address and a message address.

A call to send a message called TEXT to the output display task just described might be programmed as follows:

PL/M	8080/8085 Assembly Language

CALL RQSEND(.DISPEX,.TEXT);

```
                              LXI     B,DISPEX
                              LXI     D,TEXT
                              CALL    RQSEND
```

Typically, the exchange is associated with some other task, but it may be used by the sending task, allowing a task to send a message to itself. Note that sending a message to an exchange where a higher priority task is waiting will cause the sending task to be preempted by the receiving task.

```
WARNING
```

Once a message has been sent to an exchange, the sending task must neither send it again nor alter it in any way until it has been processed by the receiving task.

Working With Control Structures

The RMX/80 software requires certain control information in order to perform its operations. This information is provided in RMX/80 control structures located in RAM. The most important of these are Task Descriptors, Exchange Descriptors, Interrupt Exchange Descriptors, message headings, and individual task stacks. The programmer of an RMX/80 application system needs to understand these control structures, since he must initialize some of them and also access them from time to time. The following paragraphs supply a brief functional description of the control structures; a more complete discussion appears in Chapter 2 under "Details of System Operation."

Task Descriptors. A Task Descriptor is the control structure that identifies a task to the system. It indicates the priority and *status* (ready, preempted, delayed, or suspended) of the task, and gives the address of the task's stack area.

The RMX/80 Nucleus builds the Task Descriptors in user-allocated RAM from information you supply when you configure your system, or when your program requests the creation of tasks at run time.

Exchange Descriptors and Interrupt Exchange Descriptors. An Exchange Descriptor is the physical representation of an exchange in the system. It enables the RMX/80 software to keep track of all the tasks waiting at an exchange or all the messages available at the exchange. It includes pointers to a list of tasks and a list of messages. As messages are posted at the exchange, as tasks arrive to wait at the exchange, and as tasks receive messages, RMX/80 updates these lists. The RMX/80 software matches tasks and messages on a first-in/first-out basis. If, for example, several messages are queued at an exchange, the first task that waits at the exchange receives the first message from the queue, and that message is removed. The next time this task (or another task) waits at the exchange, it receives the second message from the queue.

An Interrupt Exchange Descriptor is a special type of Exchange Descriptor, provided to support the translation of hardware interrupts into messages. A separate interrupt exchange is defined for each interrupt level used in the application system. An Interrupt Exchange Descriptor contains a built-in message; a simple Exchange Descriptor does not.

The RMX/80 Nucleus builds Exchange Descriptors and Interrupt Exchange Descriptors in user-allocated RAM from information you supply when you configure your system, or when your program requests the creation of exchanges at run time.

Message Headings. Each message in the system begins with control information known as the *message heading*. It includes a pointer used to link it to other messages waiting at an exchange, indicators of length and message type, and (optionally) certain exchange address associated with the message.

A user task builds messages in some portion of RAM. This area may be dedicated to the message, or the task may request a portion of allocated memory from the Free Space Manager (see Chapter 5). The latter option is especially useful when tasks are sending variable-length messages. A message can have any length consistent with the amount of memory in the system.

Task Stacks. Each task requires its own stack so that RMX/80 can perform a context save (storing the contents of the registers, flag bytes, and program counter) whenever the task is preempted. When the task next enters the running state, the RMX/80 software restores the registers and pops the top of the stack into the program counter, so that execution resumes with the instruction that was next to be executed when the task was interrupted.

While running, the task may also store data in its stack for its own purposes. This data will not be affected by context saves as long as the stack pointer is properly updated to reflect the existence of the data on the stack. The user must, however, allocate sufficient stack space for this data and the space used by CALL instructions as well.

Specifying the System Configuration

The great degree of flexibility inherent in the RMX/80 software requires that you provide certain information to the Nucleus to enable it to initialize the system. After defining and coding your own tasks, you specify the configuration of your system by means of a section of code called the *configuration module* . You then link your configuration module to the RMX/80 tasks and user tasks you have selected. This section briefly describes the configuration module; detailed coding instructions are given in Chapter 3.

The configuration module specifies initialization structures to be located in ROM and used by the Nucleus to set up Task Descriptors and Exchange Descriptors in the system. These ROM structures are the Create Table, the Initial Task Table (which consists of Static Task Descriptors), and the Initial Exchange Table. The Nucleus uses these structures to initialize the system every time it is restarted. After tasks and exchanges are initialized, RMX/80 passes control to the highest priority task in the system.

Create Table (RQCRTB). The Create Table is simply a master table used to locate the other ROM structures needed for initialization. It consists of a pointer to the Initial Task Table, a count of the number of tasks to be initialized, a pointer to the Initial Exchange Table, and a count of the number of initial exchanges.

Initial Task Table (ITT). The Initial Task Table consists of the Static Task Descriptors for all the tasks to be created at system startup time.

Static Task Descriptors (STD's). Each task to be set up in the system at initialization must have its corresponding Static Task Descriptor, which consists of a task name, initial program counter value (starting address), stack address and length, software priority level, initial default exchange at which the task waits, and the RAM starting address of the Task Descriptor for the task. You must reserve space in RAM for a Task Descriptor corresponding to each Static Task Descriptor in the Initial Task Table.

Initial Exchange Table (IET). The Initial Exchange Table is a list of RAM addresses for all the Exchange Descriptors to be initialized by RMX/80 when the system is restarted. For each exchange address in the Initial Exchange Table, the user must reserve space at that address for the corresponding Exchange Descriptor.

Disk File System Configuration Tables. Additional information must be coded in the configuration module for a system that includes services from the RMX/80 Disk File System. Descriptions of these tables and coding instructions are supplied in Chapter 7.

General Description

The Nucleus is the heart of an RMX/80 application system and is required for all systems. To use the Nucleus, you must be familiar with a number of RMX/80's basic operating principles, which are described in Chapter 1 of this manual. These principles are not repeated in this chapter except as they apply directly to a particular operation. Unless you are already familiar with RMX/80, you should read Chapter 1 before attempting to read this or any subsequent chapter. The RMX/80 Nucleus performs the following functions:

* Initalizes the system

* Dispatches and synchronizes tasks

* Supplies timing functions

* Provides a means for controlling and exchanging data within the system.

User tasks communicate with the Nucleus by means of a set of RMX/80 operations which are defined later in this chapter.

Use Environment

The RMX/80 Nucleus and extensions are designed for use with Intel OEM computer systems. Special care has been taken to isolate hardware-dependent functions so that user tasks will execute with little or no change on different iSBC systems. However, hardware differences dictate certain software requirements. For example, the Nucleus has different versions for the iSBC 80/20, iSBC 80/30, and iSBC 80/10. The basic difference in the three versions concerns interrupt handling. The iSBC 80/20 offers eight hardware interrupt levels; the iSBC 80/30, twelve; and the iSBC 80/10, only one. The iSBC 80/10 version of the Nucleus provides a software scheme that simulates the eight interrupt levels of the iSBC 80/20.

All versions of the RMX/80 Nucleus software are designed to be fully contained on the Single Board Computer. All system hardware resources not used by the RMX/80 Nucleus are available for the user application (user tasks and user-selected RMX/80 extension tasks).

RMX/80 cannot, in general, protect itself against bugs in application tasks. One reason that this is true is that the RMX/80 data structures are maintained in RAM that is accessible to the application programs. An application task that overflows its stack or otherwise writes over an RMX/80 data structure may cause the system to malfunction in unexpected ways. The RMX/80 Debugger, described in Chapter 6, is designed to allow you to identify and correct such problems so that your application will be extremely reliable.

Memory Requirements

The RMX/80 Nucleus operations are supplied in two portions, a basic set and an optional set of operations. The optional operations delete, suspend and resume tasks and delete exchanges, and will not be needed in many user applications. The more frequently used operations are in the basic set. The Nucleus code with the basic set of operations is designed to fit into a single 2K ROM. It can be located in RAM, but this requires the system to have some means of loading the code into memory, such as the ICE-80. The basic Nucleus requires approximately 250 bytes of RAM

for data storage. If used, the optional operations require up to 210 additional bytes of storage for code, and they extend the total data storage requirements to approximately 300 bytes of RAM. Each optional operation is linked into the system only if the operation is called by a user task.

Adding tasks and exchanges to the system requires additions to the Nucleus beyond the code and/or data storage of the task itself. In addition to its own code and data storage, each task adds 17 bytes of ROM and 20 bytes of RAM to the system. Also, each task must reserve 24 bytes of stack space beyond its own needs to provide the Nucleus with space for saving the task's context should it be preempted.

Each additional exchange adds 2 bytes of ROM space and 10 bytes of RAM. Each Interrupt Exchange adds 2 bytes of ROM space and 15 bytes of RAM.

Hardware Requirements

iSBC 80/20. On the iSBC 80/20, the RMX/80 Nucleus uses the 8259 Programmable Interrupt Controller chip to provide eight priority interrupt levels. The Nucleus automatically updates the 8259 interrupt mask to block all interrupts with the same or a lower priority than the current interrupt. This also blocks all tasks with a software priority less than or equal to the current task.

The RMX/80 Nucleus uses Counter 0 of the 8253 Programmable Interval Timer chip to drive the system clock. This counter is jumpered directly to the priority 1 interrupt. Counter 0 is programmed to provide a timing interrupt every 50 milliseconds. Notice that tasks with priorities in the range 0 through 32 can block timing interrupts if their cumulative execution time exceeds 50 milliseconds. This can cause timing inaccuracies. Counter 0 and the priority 1 interrupt are normally dedicated to the timing function and are not available for other use.

Counters 1 and 2 of the 8253 are not used by the RMX/80 Nucleus; however, the Terminal Handler uses Counter 2 to generate the baud rate for the 8251 Programmable Communication Interface (USART). Since most application systems will probably include the Terminal Handler, only counter 1 of the 8253 is generally available for use by application tasks.

iSBC 80/30. All hardware requirements outlined for the iSBC 80/20 in the previous paragraphs apply also to the iSBC 80/30.

In addition, the iSBC 80/30 version supports the four on-chip interrupts provided by the 8085 processor as described under "iSBC 80/30 Interrupts" later in this chapter. The 80/30 version of the Nucleus automatically updates the 8259 and 8085 interrupt masks to block all interrupts with the same or lower priority than the current interrupt.

iSBC 80/10. The iSBC 80/10 does not have an 8259 Programmable Interrupt Controller or an on-board interval timer. The single interrupt on the 8080 processor is translated into eight interrupt levels via software. If the system requires any timing functions, a timing signal must be derived off-board to drive the RMX/80 clock logic. This imposes both hardware and software requirements, as described in Appendix G, "iSBC 80/10 Time Base Considerations."

Interfacing with the Nucleus

The user has two general interfaces with the RMX/80 Nucleus. First, the user must define the system configuration for the Nucleus. This allows the Nucleus to identify

and establish control over the various tasks and exchanges used in the system. Second, user tasks can request various services from the RMX/80 Nucleus by including calls to RMX/80 operations in the task's code.

This section briefly explains system configuration (detailed instructions for coding the configuration module are given in Chapter 3) and defines the RMX/80 Nucleus operations. In addition, four special topics—message coding, priority levels, timing, and interrupt handling—are discussed. You must have a solid understanding of these topics in order to code your application tasks.

Configuration

Before linking the system together, you must define the system configuration by supplying a Create Table and the associated Initial Task Table and Initial Exchange Table. The RMX/80 Nucleus uses this information to build the required Task Descriptors and Exchange Descriptors each time the system is restarted. (These structures are described more fully under "Details of System Operation" later in this chapter.)

A running task can alter the system configuration dynamically by creating or deleting tasks and exchanges. Run-time creation of tasks is frequently performed to simplify configuration requirements for the user. In fact, most RMX/80 extensions use this technique. For example, the Free Space Manager is comprised of a number of tasks, but you are required to supply only one Static Task Descriptor. The Free Space Manager's code contains the Static Task Descriptors for its other tasks. As a part of its initialization, the Free Space Manager generates the Task Descriptors for its other tasks, thereby "creating" the tasks. Of course, all the object code for all these tasks was previously linked into the system at the same time. The Free Space Manager creates tasks at run time only to simplify configuration requirements for you.

Run-time deletion of a task or exchange allows any RAM used by the task or exchange to be reused for some other purpose.

Nucleus Operations

Summary. The RMX/80 Nucleus operations are external procedures that may be called by your application tasks. Table 2-1 is a summary of all RMX/80 Nucleus operations.

Notice that all the operation names have the prefix RQ. This convention is true of all public and external symbols defined by RMX/80, so it is simple to avoid conflicts between RMX/80-defined and user-defined symbols.

Table 2-1. RMX/80 Nucleus Operations

NAME	OPERANDS	RETURN VALUE	FUNCTION
RQACPT	exch-addr	addr	Accept a message, if available, from specified exchange. Returns message address if available, zero otherwise.
RQCTCK	none	none	Decrement the time counters of all tasks on the Delay List, and transfer such tasks to the Ready List if their timed wait periods have expired. (Provided in iSBC 80/10 version only.)

Table 2-1. RMX/80 Nucleus Operations (Cont'd.)

NAME	OPERANDS	RETURN VALUE	FUNCTION
RQCTSK	STD-addr	none	Create task by building new Task Descriptor based on specified Static Task Descriptor.
RQCXCH	RAM-addr	none	Create exchange at specified RAM address.
RQDLVL	level	none	Disable specified interrupt level. (Has no effect in iSBC 80/10 version.)
RQDTSK	TD-addr	none	Delete task specified by Task Descriptor.
RQDXCH	exch-addr	byte	Delete specified exchange. Returns 0FFH if successful, 00H if task or message is waiting at exchange.
RQELVL	level	none	Initialize message portion of the Interrupt Exchange Descriptor associated with the specified interrupt level (the first time called only), and enable specified interrupt level. (In iSBC 80/10 version, has no effect except for the initialization.)
RQENDI	none	none	Signals end-of-interrupt in user-supplied interrupt service routine. Used in place of RQISND. (Has no effect in iSBC 80/10 version.)
RQISND	IED-addr	none	Send an interrupt message to the specified interrupt exchange. RQISND may be called from a user-supplied interrupt service routine.
RQRESM	TD-addr	none	Resume a task that has previously been suspended.
RQSEND	exch-addr, msg-addr	none	Send the message located at "msg-addr" to the exchange specified by "exch-addr."
RQSETP	addr, level	none	Allow the single-level iSBC 80/10 to simulate one of the 8 interrupt levels of the 80/20 by associating the user-supplied routine at "addr" with the specified interrupt level. Each user routine can then test one or more hardware devices to determine whether service is required. If so, it services the device and returns 0FFH. If not, the user routine returns 00H to RMX/80, which then activates the routine with the next lower priority. (Provided in iSBC 80/10 version only.)
RQSETV	addr, level	none	Set interrupt vector. Interrupts at the specified level are serviced by the user-supplied routine starting at "addr," thus bypassing RMX/80 interrupt software.
RQSUSP	TD-addr	none	Suspend execution of the task specified by the Task Descriptor.
RQWAIT	exch-addr, time-limit	addr	Wait at the specified exchange until a message is available or time limit expires. Returns address of system time-out message or user message.

Descriptions of Operations. The following descriptions supply all the information you need to use the RMX/80 operations in your application tasks.

Each description provides the format for the statement required to call the operation in PL/M. In PL/M, operations are coded differently depending upon whether the operation returns a value. When no value is returned, the operation is coded as a procedure call. When a value is returned, the operation is coded as an assignment statement or an argument of an expression (function reference). (The formats given here for operations that return a value show an assignment statement; but any of these operations may be coded either way.)

Lower-case items in the formats identify elements to be supplied by the programmer, who will generally wish to supply them in symbolic form. Specific rules for coding parameters are given in the descriptions of individual operations.

In 8080/8085 assembly language, all RMX/80 operations are coded as subroutine calls. Before calling an operation, however, the program must place any parameters required for the operation in the proper registers. The assembly language rules for passing parameters to an RMX/80 operation are the same as for passing parameters to a PL/M procedure called from an assembly language module. The first parameter is passed in the B and C registers if it is an address value, or the C register if it is a byte value. The second parameter is passed in the D and E registers if an address value, or the E register if a byte value. If the operation returns an address value, it is returned in the H and L registers. If the operation returns a byte value, it is returned in the accumulator.

In either language, you must be certain to declare the names of any operations used in a task as EXTERNAL.

The most frequently used RMX/80 operations by far are RQSEND and RQWAIT. In fact, a majority of RMX/80 systems will require only these two operations. Somewhat more complex systems may also find a need for the RQACPT operation. Therefore, these three operations are described first. The remaining operations are then described in alphabetical order.

RQSEND Operation.

Function. The RQSEND (Send Message) operation queues a message at an exchange. If one or more tasks are waiting at the exchange, the message is given to the first waiting task, and the task is made ready. In this case, both the task and the message are removed from the exchange.

Format.

      CALL RQSEND(exchange-address, message-address);

Description. RQSEND is reentrant and may be called at any time by a running task.

The RQSEND operation has two parameters. The first specifies the address of the exchange to which the message is to be sent. The second specifies the address of the message. Notice that sending a message does not imply that the message is actually moved. Instead, its address is added to the queue of message addresses at the exchange.

The sending task must not alter a message after it is sent. Since the message is not actually moved, it is in fact still accessible to the sending task. However, altering the message after it has been sent subverts the purpose of sending it to another task. This raises the question of how the sending task knows when it can reuse the RAM

to build a new message. Typically, the receiving task returns a response message to tell the sending task that the message has been processed. Note that the response message should be returned via a separate exchange. Frequently, the original message is returned as a response, possibly with updated status information or return values.

A task effectively preempts itself by sending a message to an exchange where a higher-priority task is waiting. The waiting task is readied upon receipt of the message. Since the receiving task has the higher priority, it becomes the running task. The sending task is preempted, but remains ready to run. Conversely, a task can send any number of messages to lower-priority tasks without being preempted—unless, of course, an interrupt occurs for a higher-priority task.

If a task is waiting in a timed delay when a message is received at its exchange, the delay is cancelled, and the task is made ready.

Because of the way RMX/80 messages and exchanges are structured, a message cannot be sent to more than one exchange at a time.

RQWAIT Operation.

Function. RQWAIT (Wait for Message) returns the address of a message to the calling task. The calling task waits at an exchange until a message has been received or until a specific amount of time has elapsed. If a message is received, RQWAIT returns the address of that message. If the time limit elapses before a message is received, the address returned to the calling program is the address of a system time out message (a 5-byte message with type TIMEOUTTYPE=3; see Appendix C).

Format.

    message-address=RQWAIT(exchange-address, time-limit);

Description. RQWAIT is reentrant and may be called at any time by a running task. Maximum time limit is FFFEH, or 65,534, system time units; i.e., a time limit of 65,535 or more system time units is not permitted.

The RQWAIT operation has two parameters. The first parameter specifies the address of the exchange where the task is to wait. The second specifies the time limit of the wait. A time limit value of zero indicates no time limit; i.e., the task will wait indefinitely. An RQWAIT operation with no time limit incurs less system overhead than a timed RQWAIT.

The optional time limit is specified as a number of system time units (one system time unit is 50 msec for the iSBC 80/20 or 80/30 version). A one-second wait (for the 80/20 or 80/30) is specified as 20 time units. Notice that the duration of the first time unit is usually unpredictable. Your task may enter the wait immediately after a clock tick (50 msec timer interrupt). In this case, the duration of the first time unit (for the 80/20 or 80/30) is very nearly a full 50 msec. On the other hand, your task may enter the wait an instant before a clock tick. In this case, the duration of the time unit may be nearer one millisecond. After the first clock tick, the wait becomes synchronized with the clock; all subsequent time units are a full 50 msec.

If you are using an iSBC 80/10-based system and you need timed RQWAIT operations, you must provide an off-board clock in your hardware configuration. Also note that consequently, the length of a system time unit in your 80/10 system may be other than 50 milliseconds. (See Appendix G for a discussion of timing for the iSBC 80/10.)

The exchange address parameter may be specified as zero. In this case, a default exchange address is taken from the calling task's Task Descriptor. The default exchange address may be specified in the Static Task Descriptor when the system is configured; this address is placed in the Task Descriptor at initialization. However, RMX/80 updates the exchange address of the Task Descriptor each time it encounters a RQWAIT or RQACPT operation with a non-zero exchange address. Therefore, the default exchange address in the Task Descriptor can be either the exchange address specified in the Static Task Descriptor or the address of the exchange where the task waited last.

If a message is already waiting at the exchange when the task executes the RQWAIT, the first message is removed from the Exchange Descriptor's queues, and the task remains on the Ready List. Unless an interrupt causes a higher-priority task to become ready, the calling task remains the running task.

If no message is waiting at the exchange, the calling task is removed from the list of ready tasks and added to the queue of tasks at the exchange. When a non-zero time limit is specified for the RQWAIT, the task is also queued on the Delay List. The task continues to wait at the exchange until it receives a message or until the specified time limit elapses. At this point, the task is again made ready and removed from the Delay List.

If the time limit expires before a message arrives at the exchange, RMX/80 passes the address of a system message to the task and readies it. The TYPE field of this message always contains 3H so the task can determine if it timed out.

Because of the way RMX/80 tasks and exchanges are structured, a task cannot wait at more than one exchange at a time.

RQACPT Operation.

Function. RQACPT (Accept Message) returns either the address of a message or zero to the calling task. The calling task is not required to wait. A message address is returned if a message is available at the specified exchange; zero is returned if no message is available.

Format.

message-address=RQACPT(exchange-address);

Description. RQACPT is reentrant and may be called at any time by a running task.

RQACPT has one parameter, the exchange address from which the task will accept a message, if one is available. The exchange address may be specified as zero. In this case, a default exchange address is taken from the task's Task Descriptor. This address is either the address of the exchange from which the task last took a message, or (if the task has received no messages since the system was last restarted) the default exchange address specified in the Static Task Descriptor.

RQACPT is useful when a task must handle messages from a number of different exchanges. As explained in the description of RQWAIT, a task can wait at only one exchange at a time. Furthermore, if a task attempts to wait sequentially at several exchanges, it may wait forever at one exchange even though a message is available at another exchange. In some cases the RQACPT operation can be used to overcome this problem, since it does not require the task to wait.

Executing a series of RQWAIT operations with a time limit of one time unit is approximately equivalent to executing a series of RQACPT operations, but the system overhead is slightly greater when the RQWAIT operations are used. It should be noted, however, that a task that loops through a series of RQACPTS prevents lower priority tasks from running.

Care must be taken when using the default exchange address in a task that uses both RQWAIT and RQACPT operations. Both of these operations modify the exchange address stored in the Task Descriptor for that task. Unless care is taken, the task could end up trying to accept and wait for messages at the same exchange.

If no message is currently available at the specified exchange, RQACPT returns zero to the calling task. The task should, of course, test for this condition.

RQCTCK Operation.

Function. RQCTCK (Clock Tick) decrements the time counters of all tasks on the Delay List each time another system time unit has passed, and transfers such tasks to the Ready List when their timed wait periods have expired. The RQCTCK operation is defined only for the iSBC 80/10 version; it's function is performed automatically in the iSBC 80/20 and 80/30 versions.

Format.

CALL RQCTCK;

Description. The RQCTCK procedure is not reentrant and therefore must only be called by one task at a time.

RQCTCK must be called by the polling routine that services the interrupt from the clock hardware. The user-supplied variable RQTCNT (tick count) is used to determine the number of clock interrupts that correspond to a system time unit. If you do not set this variable, RQTCNT assumes a default value of 50. This provides a system time unit of 50 msec if the clock interrupt occurs once each millisecond.

For additional information concerning iSBC 80/10 timing routines, see Appendix G.

RQCTSK Operation.

Function. RQCTSK (Create Task) identifies a task to RMX/80, initializes it, and makes it ready for execution.

Format.

CALL RQCTSK (Static Task Descriptor-address);

Description. RQCTSK is reentrant and may be called at any time by a running task.

RQCTSK has one parameter. This parameter must be the address of a Static Task Descriptor. The Static Task Descriptor may be in either ROM or RAM, but is normally in ROM. RQCTSK uses the information in the Static Task Descriptor to build a Task Descriptor in RAM. After the Task Descriptor is built, RQCTSK makes the task ready and inserts it in the Ready List according to its priority.

RQCTSK also initializes the newly created task's stack by filling the entire stack area with the value UNUSED FLAG (0C7H). This flag can be useful when testing a system. The absence of the flag indicates that the stack area has either been completely filled or has overflowed. A large number of flags in the stack area may indicate that the stack is excessively large and can be reduced.

As mentioned earlier, creating a task dynamically implies that the code for that task is already in memory or can somehow be placed into memory.

RQCXCH Operation.

Function. RQCXCH (Create Exchange) identifies and initializes an exchange for RMX/80.

Format.

CALL RQCXCH (exchange-address);

Description. RQCXCH is reentrant and may be called at any time by a running task.

RQCXCH has one parameter. This parameter must be the address of ten bytes of contiguous RAM where RMX/80 is to build a new Exchange Descriptor.

The RMX/80 Nucleus uses RQCXCH to build and initialize the exchanges specified for the system configuration in the Create Table. However, user tasks can also call RQCXCH, if desired. In this case you must be certain to create the exchange before any other task attempts to send a message to, or wait at, the exchange.

The Create Table must specify at least one Static Task Descriptor, but it may specify zero exchanges. In this case, the user must be certain to build at least one Interrupt Exchange Descriptor before the system falls into the idle task. The idle task puts the processor into a HALT. The only way to get out of a HALT (short of a system RESET) is by honoring an external interrupt. To create an Interrupt Exchange Descriptor, you must call RQCXCH (to initialize the first ten bytes), then RQELVL (to initialize the last five bytes).

RQDLVL Operation.

Function. RQDLVL (Disable Level) disables the specified interrupt level by modifying an internal system mask. The disable takes effect immediately.

Format.

CALL RQDLVL(level);

Description. RQDLVL is reentrant and may be called at any time by a running task.

The operation has one parameter, the interrupt level to be disabled. Because only one level may be specified, you must call RQDLVL once for each interrupt level to be disabled. Permissible values of "level" for the iSBC 80/20 and 80/10 versions are 0-7; for the iSBC 80/30 version, 0-7 or 9=A$LEV, 10=B$LEV, or 11=C$LEV. (A$LEV, B$LEV, and C$LEV correspond to the 8085 on-chip interrupts RST 7.5, RST 6.5, and RST 5.5, respectively.) Since the 8085 TRAP interrupt cannot be masked or unmasked, a call to RQDLVL for 8=TRAP has no effect.

Because interrupt levels in the iSBC 80/10 version are actually created by software from one hardware interrupt, it is not possible to selectively mask off specific interrupt levels with the iSBC 80/10. For this reason, RQDLVL has no effect in the 80/10 version of RMX/80. The 80/10 version of RMX/80 follows the convention of disabling all eight interrupts whenever the running task has a priority from 0 to 128, and enabling all levels when the running task's priority is from 129 to 255.

Disabling an interrupt level that is already disabled has no effect.

RQDTSK Operation.

Function. RQDTSK (Delete Task) removes the specified task from the Task List and any other list in which it appears.

Format.

CALL RQDTSK (Task Descriptor-address);

Description. RQDTSK is reentrant and may be called at any time by a running task.

RQDTSK has one parameter which must specify the address of the Task Descriptor of the task to be deleted.

If the deleted task is also the running task, RMX/80 starts execution of the current highest priority ready task upon completion of the RQDTSK operation.

Deleting a task frees up the memory required for its Task Descriptor and stack. You can re-use this memory for other purposes, if desired.

Note that a task may delete itself by executing a call to RQDTSK with the parameter RQACTV. (RQACTV is an RMX/80-defined PUBLIC variable which contains the Task Descriptor address of the running task.)

RQDTSK is an optional Nucleus operation; it is linked into a system only if it is called by a user task or specified explicitly in the LINK command.

RQDXCH Operation.

Function. RQDXCH (Delete Exchange) deletes the specified Exchange Descriptor. If the exchange is still in use, as indicated by the presence of either a message or task waiting at the exchange, it cannot be deleted; the value returned indicates whether the deletion is accomplished.

Format.

byte-variable=RQDXCH (exchange-address);

Description. RQDXCH is reentrant and may be called at any time by a running task.

You must be certain that an exchange is no longer of any use in the system before deleting it.

RQDXCH has one parameter which specifies the address of the Exchange Descriptor to be deleted.

RQDXCH returns a one-byte Boolean (true or false) value to indicate whether the deletion has succeeded. A returned value of 0FFH indicates that the Exchange Descriptor has been deleted. All zeros indicate failure. The deletion fails if there is a task or message queued at the exchange.

RQDXCH is an optional Nucleus operation; it is linked into a system only if it is called by a user task or specified explicitly in the LINK command.

RQELVL Operation.

Function. RQELVL (Enable Level) enables the specified interrupt level by modifying an internal system mask. The enable takes effect immediately. The first time RQELVL is called, it also initializes the message portion of the Interrupt Exchange Descriptor associated with this interrupt level.

Format.

CALL RQELVL(level);

Description. RQELVL is reentrant and may be called at any time from a running task.

The operation has one parameter, which must specify the interrupt level to be enabled. Because only one level may be specified, you must call RQELVL once for each interrupt level to be enabled. Permissible values of "level" for the iSBC 80/20 and 80/10 versions are 0-7; for the 80/30 version, 0-7 or 9=A$LEV, 10=B$LEV, or 11=C$LEV. (A$LEV, B$LEV, and C$LEV correspond to the 8085 on-chip interrupts RST 7.5, RST 6.5, and RST 5.5, respectively.) Since the 8085 TRAP interrupt cannot be masked off, and since no Interrupt Exchange Descriptor is defined for TRAP, a call to RQELVL for 8=TRAP has no effect.

Enabling an interrupt level does not override the software priority scheme. The interrupt will still be blocked if the running task has a priority higher than or equal to the priorities associated with the enabled level.

Since the iSBC 80/10 interrupt levels are created by software from a single hardware interrupt, specific numbered levels cannot be masked off in the 80/10 version. The RQELVL operation has no effect on hardware on the 80/10. RMX/80 follows the convention of disabling all eight interrupt levels whenever the running task has a priority from 0 to 128, and enabling all levels when the running task's priority is from 129 to 255. However, in an iSBC 80/10 system RQELVL must still be called to initialize the message portion of the Interrupt Exchange Descriptor for that level. This must be done after the call to RQSETP for the same level.

Enabling an interrupt level that is already enabled has no effect.

RQENDI Operation.

Function. RQENDI (End Interrupt) is to be used only in user-supplied interrupt routines. The operation informs RMX/80 that the user routine has completed the interrupt operation; this allows RMX/80 to perform the appropriate hardware functions required to acknowledge the interrupt. The exact functions performed depend on which Single Board Computer is being used. RQENDI is meaningless (i.e., a null procedure) in the iSBC 80/10 version.

Format.

CALL RQENDI;

Nucleus   RMX/80

Description. No parameters are required with the RQENDI operation, and RQENDI does not return any value.

Because RQENDI is used only in user-supplied interrupt service routines, it must be used in conjunction with an RQSETV (iSBC 80/20 and 80/30) operation. (Refer to the descriptions of these operations, and to the "Interrupt Handling" section later in this chapter, for further information.) User-supplied interrupt service routines operate completely outside the RMX/80 task framework. The only operations that may be called from such a routine are RQENDI and RQISND. Note that both of these routines must be called with interrupt disabled.

RQISND Operation.

Function. RQISND (Interrupt Send) is to be used only in user-supplied interrupt service routines. The RQISND operation sends a message to an interrupt exchange. RQISND also informs RMX/80 that the user routine has completed an interrupt operation; this allows RMX/80 to perform the appropriate hardware functions required to acknowledge the interrupt. The exact functions performed depend on which Single Board Computer is being used.

Format.

        CALL RQISND(Interrupt Exchange Descriptor-address);

Description. RQISND is not reentrant and must be called only when interrupts are disabled. As long as RQISND is called only from user-supplied interrupt service routines entered via an RQSETV or RQSETP operation, these requirements are met implicitly. Interrupts are disabled when a user-supplied interrupt service routine is entered, and are not enabled again until just before the routine executes the RET (return) instruction.

RQISND has one parameter, which must be the address of an Interrupt Exchange Descriptor.

Because RQISND is used only in user-supplied interrupt service routines, it must be used in conjunction with either an RQSETV (iSBC 80/20 and 80/30) or RQSETP (iSBC 80/10) operation. (Refer to the descriptions of these operations, and to the "Interrupt Handling" section later in this chapter, for further information.) User-supplied interrupt service routines operate completely outside the RMX/80 task framework. The only operations that may be called from such a routine are RQISND and RQENDI.

RQRESM Operation.

Function. RQRESM (Resume Task) allows a previously suspended task to compete once again for system resources. If the specified task is on the Suspend List, RMX/80 enters the task on the Ready List according to its priority. If this makes it the highest priority task in the system, it becomes the running task.

Format.

        CALL RQRESM(Task Descriptor-address);

Description. RQRESM is reentrant and can be called at any time by a running task.

RQRESM has a single parameter which must specify the address of the Task Descriptor for the task that is to be resumed.

RQRESM clears the suspended flag in the STATUS field of the Task Descriptor. Clearing this flag for a task that is not currently suspended has no effect.

The RQRESM and RQSUSP (Suspend Task) operations are provided mainly for debugging purposes. They are optional operations that are linked into the system only if called by a task. If your system does not call one of these operations but you wish to use the operation for debugging, you must link it in separately, as described under "Linking and Locating" in Chapter 3.

RQSETP Operation.

Function. By associating a user-supplied interrupt polling routine with an interrupt level number, RQSETP (Set Poll) enables the iSBC 80/10 to simulate the eight interrupt levels of the iSBC 80/20. The RQSETP operation is defined only for the iSBC 80/10 version.

Format.

    CALL RQSETP(procedure-address,level-number);

Description. All interrupts on the iSBC 80/10 are received through hardware interrupt level 7. RQSETP allows the user to define software pseudo-interrupt levels 0 through 7, where 0 is the highest. When an interrupt is received, RMX/80 automatically calls the set of user-supplied interrupt polling routines one at a time, starting with level 0 and continuing through level 7. Thus, the system polls up to eight possible levels of interrupts. The user must supply an interrupt handling routine for each RQSETP operation.

It is not necessary to supply polling routines for unused interrupt levels, since RMX/80 supplies default routines in UNRSLV.LIB. (These are null procedures.) The compiler warning messages generated when these routines are not declared may be ignored. However, you may wish to declare polling routines for the unused levels in order to avoid the warning messages.

User routines named in RQSETP operations have a unique status. They are not tasks, but procedures or subroutines. Nor do they operate completely outside the RMX/80 software framework. These routines are invoked by the RMX/80 polling software, which calls the routines in order by their associated interrupt levels.

When an interrupt occurs, control passes to an RMX/80 routine that controls the order in which the user-supplied polling routines are executed. This RMX/80 routine performs a context save, but does not supply a stack for the user polling routine. Therefore, the user-supplied routines share the stack of the task executing when the interrupt occurs. Since interrupts cannot be predicted, *each task must include enough additional stack space to accommodate its own needs and the needs of the interrupt polling routines.* Rather than impose this burden on all tasks, the user can reserve a block of RAM as the stack for the polling routines. When invoked, each polling routine can save the current Stack Pointer and substitute its own. The user routine must restore the Stack Pointer before returning to the RMX/80 polling controller or calling RQISND. This technique is not required if the user-supplied routine pushes no data on the stack and calls no procedure other than RQISND.

Each user-supplied polling routine must inform the RMX/80 Nucleus whether it assumes responsibility for the interrupt that has occurred. It does this by returning a value to RMX/80. For PL/M programmers, user-supplied interrupts are byte-type procedures. The procedure must return a TRUE value (0FFH) if it accepts responsibility for the interrupt; the procedure returns FALSE (00H) if it does not accept

responsibility for the interrupt. For assembly-language programmers, these interrupt routines are subroutines called by RMX/80. Before executing a return instruction, the subroutine must place either 0FFH or 00H in the accumulator.

For additional information concerning interrupts, see "Interrupt Handling" later in this chapter.

RQSETV Operation.

Function. RQSETV (Set Interrupt Vector) allows the user to bypass the normal RMX/80 interrupt handling software and process interrupts directly. Although RQSETV is permitted for any version of the RMX/80 Nucleus, it is intended primarily for the iSBC 80/20 or 80/30. Using this operation with the iSBC 80/10 version causes the software interrupt polling scheme to become inoperative, rendering the RQSETP operation meaningless.

Format.

CALL RQSETV(procedure-address, level);

Description. RMX/80 normally vectors (directs) interrupts to its own interrupt handlers. These typically cause a context switch to a user task waiting at an interrupt exchange. In some cases this may not be fast enough to service high-speed interrupts. RQSETV allows the user to direct interrupts to his own interrupt routine rather than to RMX/80.

RQSETV requires two parameters. The first parameter must specify the address of the user-supplied interrupt routine. The second parameter specifies the interrupt level for which the user supplies his own software. Permissible values of "level" for the iSBC 80/20 version are 0-7; for the iSBC 80/10, level 7; for the iSBC 80/30, 0-7 or 8=TRAP, 9=A$LEV, 10=B$LEV, or 11=C$LEV. (TRAP, A$LEV, B$LEV, and C$LEV correspond to the 8085 on-chip interrupts TRAP, RST 7.5, RST 6.5, and RST 5.5, respectively.)

The user-supplied interrupt routine operates outside the task framework provided by RMX/80. Control passes directly to the user-supplied routine; no context save is performed. In effect, the user-supplied routine just rides on top of the task executing when the interrupt occurs. Therefore, the user-supplied routine must save and restore any registers it uses. Notice that the user-supplied routine does not have its own stack and therefore shares the stack that was in use when the interrupt occurred. Since the occurrence of an interrupt cannot be predicted, each task must include enough additional stack space to accommodate its own needs and the needs of the interrupt routine. Rather than impose this burden on all tasks, the user procedure can reserve RAM for its own stack. When invoked, the procedure can save the current Stack Pointer and substitute its own. The procedure must restore the Stack Pointer before it returns to the interrupted task or calls RQISND. Allocating a separate stack is not necessary if the procedure does not push anything on the stack and does not call any procedures other than RQISND and RQENDI.

Note that if the interrupt procedure is coded as a PL/M procedure with the INTERRUPT attribute, the primary control $NOINTVECTOR must be specified at the beginning of the module in which it is contained. This is necessary to prevent the compiler from attempting to write a restart vector over ROM used by RMX/80 code.

RQSETV is provided because RMX/80 imposes a certain amount of overhead when it services interrupts. In addition to performing a context save, RMX/80 also updates several of its control structures. The user may be able to bypass some of the overhead by supplying his own interrupt handling routing entered via RQSETV. For an example, see "User-Supplied Interrupt Routines" later in this chapter.

Most applications require only one RQSETV operation in the system for each level of interrupt for which the user supplies an interrupt routine. It is possible, however, to change peripheral devices while the system is running. In this case, RQSETV might be used to change the interrupt handling to service different devices using the same Interrupt Exchange Descriptor. Note that one cannot reset the interrupt vectors to the standard system interrupt service routines once they have been redirected with RQSETV.

For additional information concerning interrupts, refer to "Interrupt Handling" later in this chapter.

RQSUSP Operation.

Function. RQSUSP (Suspend Task) suspends a task, thus making it unable to compete for system resources.

Format.

    CALL RQSUSP (Task Descriptor-address);

Description. RQSUSP is reentrant and may be called at any time by a running task.

RQSUSP has one parameter which must specify the address of the Task Descriptor for the task that is to be suspended.

RQSUSP alters the STATUS field of the Task Descriptor to indicate that the task is suspended. Subsequent actions depend on whether the specified task is running, ready, or waiting:

- A running task may suspend itself. The running task surrenders control of the processor, and its Task Descriptor is placed on the Suspend List. Control of the processor passes to the highest priority ready task. When the suspended task is resumed and gains control of the processor, execution resumes at the next instruction after the call for RQSUSP.

- When a ready task is suspended, its STATUS is set to suspended and its Task Descriptor is placed on the Suspend List.

- When a waiting task is suspended, its STATUS is set to suspended. However, no further action is taken until the task becomes ready. The Task Descriptor is placed on the Suspend List when the task becomes ready. This ensures that the task is ready to be executed when it is resumed.

Suspending a task that is already suspended has no effect.

RQSUSP is not required in most systems, since you can achieve the same effect by having the task wait at an exchange. In effect, the task remains suspended until a message is sent to that exchange. However, suspending tasks during debugging is useful, since it simplifies the system and allows you to concentrate on particular portions of the system.

The RQSUSP and RQRESM (resume) operations are provided mainly for debugging purposes. If your system does not call one of these operations but you wish to use the operation for debugging, you must link it in separately, as described under "Linking and Locating" in Chapter 3.

Message Coding

General Rules. Before sending a message via the Nucleus operation RQSEND, a user task must build the message in RAM. All RMX/80 messages must conform to certain structural rules, whether the receiving task is an RMX/80 extension task or a user task. These rules are given in the following paragraphs.

Every message consists of a heading which may be from five to nine bytes long, and an optional remainder which may be any length. The generalized format of an RMX/80 message is shown in figure 2-1. RMX/80 supplies the data for the first two bytes of the message (the LINK field); the user tasks supply the remaining fields, including the remainder. The remainder and the HOME EXCHANGE and RESPONSE EXCHANGE fields are optional, in that they are not required in all messages; however, certain RMX/80 extension tasks receive user messages that must include one or more of these fields. Note that the optional fields must start at the offsets shown in figure 2-1. Therefore, in a message that uses RESPONSE EXCHANGE but not HOME EXCHANGE, RESPONSE EXCHANGE must still be located at byte 7. (In this case, the two bytes or "filler" (bytes 5 and 6) could be used for any user data.)

RMX/80 uses the LINK field to link together multiple messages waiting at the same exchange. This field should not be changed by user tasks.

The user task must set the LENGTH field. The value set must be the length in bytes of the entire message, including the remainder.

TYPE is a one-byte field whose meaning is defined by the user or by RMX/80. Type numbers 0 through 63 are reserved for RMX/80-defined message types. Some of these message types are used internally by the Nucleus, and some communicate information to RMX/80 extension tasks. A list of all currently defined RMX/80 message types is given in Appendix C. (For convenience in PL/M programming, RMX/80 supplies INCLUDE files that assign symbolic names to these message types. The names of these INCLUDE files are given in Appendix A.) Type numbers 64 through 255 are available for user-defined message types.

The HOME EXCHANGE field contains the address of the exchange to which the message is to be sent when it has no further use in the system. This field is useful for managing pools of messages.

The RESPONSE EXCHANGE field contains the address of the exchange to which a logical response to this message should be sent. This information is often very useful to the programmer. Assume, for example, that task A builds a data string and sends it to task B for processing, and that task A cannot proceed until the string is processed. Task A can wait at a response exchange immediately after sending a message to task B. In turn, task B can send a message to this response exchange when it has completed processing the data string.
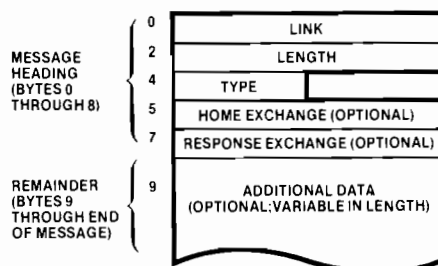
Figure 2-1. RMX/80 Message Format

Messages to RMX/80 Extension Tasks. RMX/80 extension tasks that receive messages from user tasks generally have more specific rules for the coding of these messages. The chapters on the RMX/80 extension tasks supply format diagrams and rules for these particular types of messages.

In messages used to communicate with extension tasks, some fields may be designated as unused. The application task may use these fields for its own purposes.

Priority Levels

As described in Chapter 1, RMX/80 uses a system of priority levels to determine scheduling of tasks. You must assign a priority level number to every RMX/80 extension task and user task in your application system.

When a higher-priority task preempts execution of a lower-priority task (either as the result of an interrupt or by timing out on a timed wait), RMX/80 saves all relevant information about the preempted task (i.e., CPU registers, flag byte, and program counter) on the task's stack so that it can eventually resume execution as though it were never interrupted. This process is known as a context save. In contrast, RMX/80 saves only the program counter when a task voluntarily surrenders control of the processor. (This occurs when the task sends a message to an exchange where a higher-priority task is waiting, or when it waits at an exchange where no message is queued.) Then RMX/80 simply begins execution of the task with the highest current priority.

The RMX/80 software provides 256 priority levels. Priority level zero is the highest priority; 255 is the lowest priority. It is important to remember that *the lower the priority level number, the higher the priority.*

There is nothing wrong with assigning two tasks the same priority level; in fact, in some cases it may serve a purpose to do so. For instance, it may be advisable to assign the Free Space Manager the same priority as some time-critical task that depends upon RAM allocations. When two tasks have the same priority, note that it cannot be predicted which task will be ahead of the other on the Ready List.

Priorities Associated with Interrupts. Certain software priorities correspond to one of the hardware interrupt levels provided by the iSBC 80/20 or the iSBC 80/30. A similar correspondence exists between software interrupt levels on the iSBC 80/10. These correspondences are summarized in table 2-2.

A task that is to service a particular interrupt level must be assigned a software priority in the range associated with that interrupt. For example, if an analog-digital interface uses interrupt level 2, the task associated with that interrupt must have a software priority in the range from 33 to 48.

Non-Interrupt Priority Levels. Priority levels 129 through 254 are not associated with interrupt levels. These are the levels generally used for non-interrupt-driven tasks — i.e., tasks initiated by messages sent from other tasks rather than by interrupt messages. An example of such a task is a long arithmetic routine that receives input from an interrupt-driven task. Such routines should normally be assigned lower priority levels (i.e., higher priority level numbers) than the tasks that service interrupts.

Table 2-2. Correspondence between Interrupt Levels
and Software Priority Levels

Software Priority Level	iSBC 80/20 Hardware Interrupt Level or iSBC 80/10 Pseudo-Interrupt Level	iSBC 80/30 Hardware Interrupt Level*
0**	0	A$LEV (level 7.5)
1-4	0	A$LEV (level 7.5)
5-8	0	B$LEV (level 6.5)
9-12	0	C$LEV (level 5.5)
13-16	0	0 ·
17-32	1	1
33-48	2	2
49-64	3	3
65-80	4	4
81-96	5	5
97-112	6	6
113-128	7	7
129-255	None	None

*   Note: The TRAP interrupt level on the 8085 processor cannot be masked off, and therefore transcends the RMX/80 priority scheme.

**   Software priority level 0 is used by the RMX/80 Debugger for the scan and exchange breakpoint facilities. If you plan to use these facilities, you must avoid assigning any task a priority level of 0.

The Idle Task. Priority level 255 is reserved for the idle task, which is invoked by RMX/80 when no other task in the system has a need for the processor. The idle task puts the processor in the HALT state until it receives an external interrupt. Because no instructions are fetched during a HALT, the system bus is freed for more efficient direct-memory-access data transfers.

Timing

The one facility for clock timing in an RMX/80 system is the timed RQWAIT operation. Its purpose is to allow the programmer, if he so desires, to ensure that a task will not wait at an exchange for longer that a specified length of time. The minimum interval of real time recognized by the system is a *system time unit* , which is equal to 50 msec in the iSBC 80/20 or 80/30 version. (In the iSBC 80/10 version the length of a system time unit may have some other value, as discussed in Appendix G. Using 50 milliseconds is recommended, however, so that tasks can run on alternate systems.)

It is important to note that although the system clock (Counter 0 of the Intel 8253 Programmable Interval Timer chip on the iSBC 80/20 or 80/30, or a user-supplied clock for the iSBC 80/10) may operate continuously, the clock ticks (clock interrupts) are recognized by RMX/80 only during those periods when at least one task in the system is waiting with a time delay. When no task is in a timed wait (i.e., when the Delay List is empty), the clock is "turned off" as far as RMX/80 is concerned, in order to save the system overhead time required to recognize the clock interrupts. If you wish to keep continuous track of time in your system, you must program your tasks so that one task is always in a timed wait when the clock ticks. Appendix J provides an example of a system programmed in this way.

Note also that, as explained in the description of the RQWAIT operation, the duration of the first time unit in a delay is usually unpredictable. This is because the task may enter the waiting state at any time between one clock tick and the next.

The iSBC 80/10 provides no on-board clock; so if your 80/10 system needs timed RQWAIT operations, you must furnish other clock hardware in your system. Certain programming considerations apply in interfacing clock hardware other than the 8253 with RMX/80. These are covered in Appendix G, "iSBC 80/10 Time Base Considerations."

Interrupt Handling

Interrupt handling on the iSBC 80/20 or 80/30 is a service provided by RMX/80. Interrupt handling on the iSBC 80/10, or on the iSBC 80/20 or 80/30 when user- supplied interrupt service routines are required for especially response-time-critical applications, is somewhat more complex.

RMX/80 supports all eight hardware interrupt levels on the iSBC 80/20, and all twelve levels (the eight supplied by the 8259 Programmable Interrupt Controller chip, plus the four additional levels provided on the 8085 processor) on the iSBC 80/30. (The 80/30 TRAP interrupt is a special case; see discussion under "iSBC 80/30 Interrupts.") The single-level hardware interrupt on the iSBC 80/10 is transformed via RMX/80 software into eight software interrupt levels, which RMX/80 treats as if they were the corresponding 80/20 hardware interrupts.

The following section on iSBC 80/20 interrupts describes the use of RMX/80's interrupt handling facilities, as well as the facilities provided for writing your own interrupt service routines. Much of this material is also true for the iSBC 80/30 and/or 80/10 versions of RMX/80, so all users should read this section. (80/10 users may disregard the subsection on user-supplied routines.) The sections on iSBC 80/30 and iSBC 80/10 interrupts describe only the differences from the iSBC 80/20 version.

iSBC 80/20 Interrupts. A device generates an interrupt to indicate that it is ready to send data to the system or that it is ready to accept data from the system. In the case of DMA (Direct Memory Access) devices, the interrupt signals completion of the data transfer.

A special set of interrupt exchanges is reserved for use with the eight hardware interrupt levels. By convention, these exchanges (and their corresponding Interrupt Exchange Descriptors) are named RQL0EX, RQL1EX, RQL2EX,...RQL7EX. Thus, a priority 48 task communicates with level 2 interrupts (see table 2-2) via RQL2EX. Rather than consume the memory required to predefine all eight interrupt exchanges, RMX/80 requires the user to define the interrupt exchanges used in his tasks. Like other exchanges, these may be defined either in the configuration module or at run time via RQCXCH operations. When an interrupt exchange is defined at run time, an RQELVL operation is also required to initialize the five-byte message part of the Interrupt Exchange Descriptor.

Certain interrupt levels—and their associated interrupt exchanges—have been assigned to particular RMX/80 features. RQL1EX is normally reserved for the system clock. When the Terminal Handler is included in a system, it requires RQL6EX for input and RQL7EX for output.

For each user-declared interrupt exchange, you must provide a task that waits at the exchange. This task must have a priority that corresponds to the hardware priority of the interrupt. For example, a task that waits at RQL2EX must have a priority in the range 33 through 48. The purpose of this task is to perform the required input or output operation after an interrupt for its exchange has been recognized.

RMX/80 Interrupt Handling. After system initialization, when RMX/80 passes control to the highest priority task, all hardware interrupt levels on the iSBC 80/20 are disabled. The levels required by the application tasks must be enabled via the RQELVL operation described earlier in this chapter.

When an interrupt is recognized, the following sequence of events occurs:

1. The 8080 processor disables all interrupts.

2. A context save is performed, i.e., the contents of the registers are pushed onto the stack.

3. RMX/80 performs the hardware operations required to acknowledge the interrupt.

4. RMX/80 then sends an interrupt message to the interrupt exchange associated with this interrupt level. (The interrupt message sent is the message that makes up the last five bytes of the Interrupt Exchange Descriptor.) If there is no message already waiting at this exchange, the TYPE field of the interrupt message is set to 01H, INTMSGTYPE. If there is a message already waiting at this exchange — generally indicating that an interrupt at this level has occurred previously but has not yet been serviced — the TYPE field of the interrupt message already waiting at the exchange is set to 02H, MISSEDINTTYPE, and no new message is queued at the exchange.

5. If a task is waiting at the interrupt exchange, this task is immediately made the running task (i.e., is put at the head of the Ready List), because its priority corresponds with that interrupt level and is thus necessarily higher (lower in number) than the task that was interrupted. Before passing control to the user task, RMX/80 enables interrupts with a higher priority than the current interrupt.

6. The user task performs any input or output operation associated with the interrupt and any other required processing. Input/output operations are performed through the microprocessor's I/O ports or through memory locations. (I/O port assignments are part of the system's hardware design.) The user task must not return the interrupt message.

7. When the user task surrenders control by again waiting at the interrupt exchange, RMX/80 continues execution of the ready task with the highest current priority and resets the interrupt mask to correspond with the priority of that task. If the task has priority level 129 or lower (priority level number 129-255), all interrupts are enabled.

Note that while servicing an interrupt, RMX/80 masks off interrupt levels equal to or lower (higher in number) than the interrupt being serviced. Thus, while a level 2 interrupt is being serviced, hardware interrupt levels 2 through 7 are temporarily blocked from causing interrupt messages to be sent. However, the interrupt hardware is still able to set latches to record the occurrence of subsequent interrupts so that the interrupts can be received when they become unmasked.

When you let the iSBC 80/20 version of the RMX/80 Nucleus manage the interrupt system for you, all you must supply is a user task or tasks to wait at the appropriate interrupt exchange. Typically, this task performs any· input or output associated with the interrupt.

As a general rule, tasks that wait at interrupt exchanges should be as brief as possible so that subsequent interrupts can be enabled. When an interrupt generates a great deal of processing, it is useful to divide the work among two or more tasks. The task that waits at the interrupt exchange can service the interrupt and send any necessary data to a task with a priority of 129-254. This second task can then process the data. This technique allows the enabling of subsequent interrupts as quickly as possible.

Notice that user tasks wait at an interrupt exchange; they very seldom send a message to an interrupt exchange, even though RMX/80 does not prohibit such an operation. Sending a message to an interrupt exchange can make the receiving task think an interrupt has occurred when one has not. (This can be overcome by setting the interrupt message's TYPE field to a type other than INTMSGTYPE(1) or

MISSEDINTTYPE(2).) Also, sending messages via an interrupt exchange might overload the receiving task and cause missed interrupts. (This is, however, a useful debugging technique.)

Missed Interrupts. It is important to make clear that there are two different ways in which an interrupt arriving at the iSBC 80/20 from an external device can be "missed," or in other words, not translated into an interrupt message and allowed to wake up an interrupt task waiting at an interrupt exchange.

In the first case, which is discussed in the previous section, the interrupt is received, but an interrupt message has already been sent to the exchange and not yet received by a task. In this case the TYPE field of the first interrupt message already queued at the exchange is changed to MISSEDINTTYPE(2), and the second interrupt message is not queued at the exchange. It is the responsibility of the user application to check for this kind of missed interrupt by testing the TYPE field of the message portion of the Interrupt Exchange Descriptor. Note that the RMX/80 software provides an indication of the first missed interrupt message at a particular level, but does not distinguish between a single missed interrupt and a succession of missed interrupts.

In the second case, the interrupt arrives at the appropriate hardware interrupt pin on the iSBC 80 (for the iSBC 80/20, this will be one of the interrupt pins on the 8259 chip), but is held up there because the associated interrupt level is currently masked off by RMX/80. In this case, the interrupt signal is not translated into an interrupt message until RMX/80 unmasks that interrupt level. If a second interrupt occurs before the level is unmasked (acknowledged), no record of the first will remain.

Note that the second case will always hold true when a second interrupt signal arrives while the task designated to service that interrupt level is running in response to an earlier occurrence of the interrupt. (The fact that the interrupt service task is running means that all interrupts of that level or lower are masked off.) The second case will also hold true if a task servicing a higher-priority interrupt is running. The first case may be true only if the task servicing the interrupt level in question is waiting at an exchange other than the interrupt exchange, and a lower-priority task is currently running. In this situation the interrupt level will not be masked off and interrupt signals arriving at that level will be translated by RMX/80 into an interrupt message, which will be queued at the interrupt exchange until the interrupt task is made running again.

In defining and coding your application tasks for servicing interrupts, it is important for you to be aware of the possibilities discussed in the preceding paragraphs. There are ways in which you can minimize the chances of missing interrupts, while keeping your application efficient. For instance, if a task servicing a high-priority interrupt (e.g., for terminal input) must also wait at another (non-interrupt) exchange, you could split this task into two separate tasks—one to wait at the interrupt exchange and store incoming data in a buffer, the other to wait at the other exchange and process the buffer as it is able to.

User-Supplied Interrupt Service Routines. User-supplied interrupt service routines impose a substantial programming burden. These routines are required only when the interrupt handling scheme provided by the RMX/80 Nucleus is unable to respond quickly enough in the user's environment.

The iSBC 80/20 and 80/30 versions of RMX/80 provide three operations to support user-supplied interrupt routines: RQSETV, RQISND, and RQENDI.

To make use of these facilities, you should set up your interrupt service routine with a call to RQSETV, which substitutes the address of a user-supplied PL/M procedure or assembly-language subroutine in place of RMX/80's normal interrupt-handling software. When an interrupt for which an RQSETV opera-

tion has been given is recognized, control passes immediately to the specified PL/M procedure or assembly-language subroutine. (In effect, RQSETV substitutes a CALL instruction for the normal RMX/80 software.) This has several programming implications:

• Because the normal RMX/80 software is bypassed, the user-supplied routine receives control without benefit of any software services. The interrupt mask remains unchanged, and no context save for the executing task is performed. This requires the user-supplied task to save and restore any registers modified by the interrupt service routine. (The PL/M compiler will generate code to save and restore the registers if your interrupt routine is declared an INTERRUPT procedure. If you use an INTERRUPT procedure, note that you must specify the primary control $NOINTVECTOR at the beginning of the module.)

• Under no circumstances should an EI (Enable Interrupts) instruction be executed within the user-supplied routine. This can cause a loss of system integrity. However, the user-supplied routine must enable interrupts when it returns control to the task preempted by this interrupt. These seemingly conflicting requirements can be met because of a feature of the EI instruction. The effect of the Enable Interrupts instruction is delayed until the next sequential instruction has been executed. Therefore, the instruction sequence EI, RET (return) meets the requirement of not executing an Enable Interrupts within the user-supplied routine, but also enables interrupts before returning to the preempted task. (Note that the code generated by the PL/M compiler for an INTERRUPT procedure satisfies this requirement.)

• The user-supplied routine is either a PL/M procedure or an assembly-language subroutine operating completely outside the normal RMX/80 task framework. Therefore, the routine has no particular priority or environment of its own. The routine must draw its resources from whatever task was running when the interrupt occurred. Any registers and stack space used by the routine are taken from that task. Since the occurrence of an interrupt cannot be predicted, all tasks in the system must reserve enough stack space for their own needs, 24 bytes for an RMX/80 context save, and sufficient stack space for the needs of the user-supplied interrupt service routine(s). User-supplied routines that use excessive amounts of stack space may cause stack overflows in the Terminal Handler and/or Free Space Manager. Alternatively, the user-supplied routine can create its own stack. This is done by reserving the required space in RAM. When the user-supplied routine begins execution, it saves the current Stack Pointer and replaces it with the address of its own stack. The routine must restore the Stack Pointer before returning control to the preempted task or calling RQISND, whichever is done first.

• When the user-supplied routine begins execution, the top of the stack contains the return address needed to resume execution of the task running when the interrupt was recognized. The user-supplied interrupt needs this address to return to the interrupted task.

Because the user-supplied routine operates completely outside the RMX/80 task framework, the normal RMX/80 operations cannot be used. RMX/80 provides the RQENDI and RQISND operations to overcome this limitation. These are the only RMX/80 operations that should be called from within a user-supplied interrupt service routine.

RQENDI and RQISND both signal the RMX/80 software that the interrupt has been serviced. RMX/80 then acknowledges the interrupt. RQISND has the extra function of sending a message to the specified interrupt exchange. This has the effect of starting execution of any task waiting at that interrupt exchange, thereby reentering the RMX/80 software environment. System conventions require that the task invoked by RQISND run at a priority level associated with the interrupt exchange. Remember that all lower-priority tasks are still blocked from executing by

the RMX/80 software. Therefore, interrupts with a priority equal to or lower than the task invoked by the RQISND are disabled. RMX/80 alters this condition only when the task invoked by the RQISND surrenders control of the processor.

As mentioned previously, RQENDI acknowledges the hardware interrupts on the iSBC 80. It is particularly useful when the user-supplied interrupt routine must handle a burst of interrupts. Typically, such a user-supplied routine handles a whole string of data before calling RQISND.

The flowchart of figure 2-2 illustrates the interaction between a user-supplied interrupt service routine and a task that waits at an interrupt exchange. The user-supplied routine fills a buffer with input data. When the buffer is full, the user-supplied routine wakes up a task by sending an interrupt message to an interrupt exchange via an RQISND operation.

Because the interrupt task has a priority that corresponds to the hardware interrupt that invoked the RQSETV interrupt routine, further interrupts at this level are blocked until the interrupt task again waits at the interrupt exchange. Therefore, the buffer can be processed and re-initialized before another interrupt can alter the buffer. In many cases, processing the buffer may consist only of sending the data to a lower priority task. Note that all application processing code in the RQSETV routine must precede the RQENDI or RQISND operation.

The following PL/M listing illustrates a typical user-supplied interrupt service routine for the iSBC 80/20, following the example flowcharted in figure 2-2. Note that since a PL/M INTERRUPT procedure is used, the primary control $NOINT-VECTOR must be specified at the beginning of the module, to prevent the compiler from attempting to write a restart vector over ROM used by RMX/80 code.
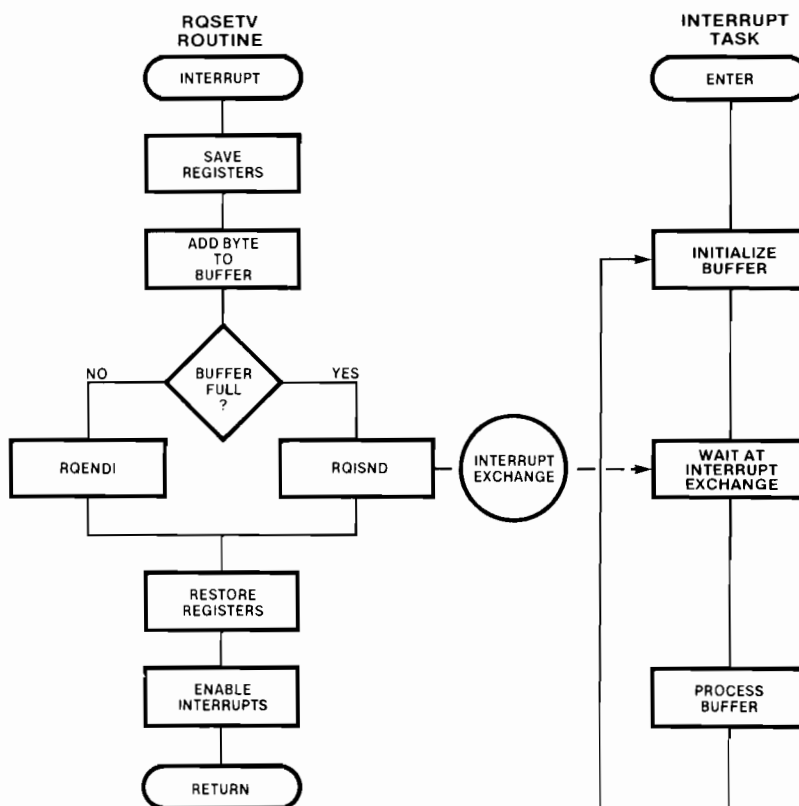
Figure 2-2. Buffered Input via User-Supplied Interrupt Service Routine

```
$NOINTVECTOR
SERVICE:
DO;

/* EXTERNALS */

DECLARE DATA$BUFFER(1) BYTE EXTERNAL;
DECLARE DATA$PNTR ADDRESS EXTERNAL;
DECLARE DATA$COUNT ADDRESS EXTERNAL;

RQISND:
    PROCEDURE(IED$ADDRESS) EXTERNAL;
        DECLARE IED$ADDRESS ADDRESS;
    END RQISND;

RQENDI:
    PROCEDURE EXTERNAL;
    END RQENDI;

/* PUBLICS */

DECLARE RQL7EX(15) BYTE PUBLIC;

INT$SVC:
    PROCEDURE INTERRUPT 7;

        DECLARE DATA$PORT LITERALLY '0ECH';

        DATA$BUFFER(DATA$PNTR) = INPUT(DATA$PORT);
        DATA$PNTR = DATA$PNTR + 1;
        IF DATA$PNTR >= DATA$COUNT THEN
            CALL RQISND(.RQL7EX);
        ELSE
            CALL RQENDI;

    END INT$SVC;

END SERVICE;

BLOCK$READ:
DO;

/* PUBLICS */

DECLARE FOREVER LITERALLY 'WHILE 1';
DECLARE DATA$BUFFER(256) BYTE PUBLIC;
DECLARE DATA$PNTR ADDRESS PUBLIC;
DECLARE DATA$COUNT ADDRESS PUBLIC;
DECLARE REQ$EXC EXCHANGE$DESCRIPTOR PUBLIC;

/* EXTERNALS */

DECLARE RQL7EX(15) BYTE EXTERNAL;

INT$SVC:
    PROCEDURE EXTERNAL;
    END INT$SVC;

RQSETV:
    PROCEDURE(SERVICE$ADDRESS,INT$LEVEL) EXTERNAL;
```

```
                    DECLARE SERVICE$ADDRESS ADDRESS,
                            INT$LEVEL BYTE;
                END RQSETV;
                RQELVL:
                    PROCEDURE(INT$LEVEL) EXTERNAL;
                        DECLARE INT$LEVEL BYTE;
                    END RQELVL;

                RQDLVL:
                    PROCEDURE(INT$LEVEL) EXTERNAL;
                        DECLARE INT$LEVEL BYTE;
                    END RQDLVL;

                RQSEND:
                    PROCEDURE(EXCHANGE$ADDRESS,MESSAGE$ADDRESS) EXTERNAL;
                        DECLARE(EXCHANGE$ADDRESS,MESSAGE$ADDRESS) ADDRESS;
                    END RQSEND;

                RQWAIT:
                    PROCEDURE(EXCHANGE$ADDRESS,MESSAGE$ADDRESS)
                                                         ADDRESS EXTERNAL;
                        DECLARE(EXCHANGE$ADDRESS,MESSAGE$ADDRESS) ADDRESS;
                    END RQWAIT;

                USART$READER:  /*MUST BE A TASK WITH PRIORITY 113-128*/
                    PROCEDURE PUBLIC;
                        DECLARE MSG$PNTR ADDRESS;
                        DECLARE MSG BASED MSG$PNTR STRUCTURE(
                                                LINK ADDRESS,
                                                LENGTH ADDRESS,
                                                TYPE BYTE,
                                                HOME$EXCHANGE ADDRESS,
                                                RESPONSE$EXCHANGE ADDRESS,
                                                ADDR ADDRESS);
                        DECLARE INTR$MSG ADDRESS;

                        CALL RQSETV(.INT$SVC,7);

                        DO FOREVER;
                            MSG$PNTR = RQWAIT(.REQ$EXC,0);
                            DATA$PNTR = 0;
                            DATA$COUNT = 256;
                            CALL RQELVL(7);
                            INTR$MSG = RQWAIT(.RQL7EX,0);
                            CALL RQDLVL(7);
                            MSG.ADDR = .DATA$BUFFER;
                            CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSG$PNTR);

                        END; /* OF DO FOREVER */
                    END USART$READER;

                END BLOCK$READ;
```

iSBC 80/30 Interrupts. The iSBC 80/30 version of RMX/80 processes interrupts
in essentially the same way as the iSBC 80/20 version. The differences lie in
RMX/80's support of the additional interrupt levels available on the 8085 processor
chip on the iSBC 80/30. These levels (TRAP, RST 7.5, RST 6.5, and RST 5.5) are
recognized by RMX/80 as byte values 8, 9, 10, and 11, respectively; the PL/M IN-
CLUDE file INTRPT.ELT assigns them the symbolic values "TRAP," "A$LEV,"
"B$LEV," and "C$LEV," respectively. The priorities of these interrupts are higher

than the levels 0-7 provided on the 8259 Programmable Interrupt Controller chip. (For a list of the software priorities corresponding to each of the iSBC 80/30 interrupts, refer to table 2-2.)

RMX/80 Interrupt Handling. The following differences (from the 80/20 version) apply to the iSBC 80/30 version:

* The 80/30 version of RMX/80 defines eleven interrupt exchanges: RQLAEX, RQLBEX, RQLCEX, and RQL0EX through RQL7EX. No interrupt exchange is associated with the TRAP interrupt; the special case of this interrupt is discussed later in this section.

* The RQELVL and RQDLVL operations recognize the additional interrupt levels available on the iSBC 80/30. (However, since the TRAP interrupt cannot be masked off, use of the TRAP level parameter (8) with RQELVL or RQDLVL has no effect.)

User-Supplied Interrupt Service Routines. The facilities provided by the iSBC 80/30 for handling user-supplied interrupt service routines are identical to those of the iSBC 80/20, with two exceptions:

* The RQISND operation recognizes the address of any of the eleven Interrupt Exchange Descriptors defined for the iSBC 80/30.

* The RQSETV operation recognizes all twelve interrupt levels available on the iSBC 80/30. (However, use of the TRAP interrupt is subject to special cautions, which are discussed in the following paragraphs.)

Notes on 8085 Hardware Interrupts. Note that:

* The TRAP interrupt (TRAP pin on the 8085) is edge/level-triggered. The interrupting signal must go high and stay high to trigger an interrupt.

* The A$LEV interrupt (RST 7.5 pin on the 8085) is edge-triggered. A pulse on this line will set an internal flip-flop. The 8085 automatically resets this flip-flop when it passes control to the level 7.5 interrupt service routine. The flip-flop will be set (interrupt pending) even when the A$LEV interrupt is masked out (using a call to RQDLVL), and the interrupt will be acknowledged if and when A$LEV is unmasked and re-enabled. (This pending interrupt may optionally be eliminated by a user task using an assembly language SIM instruction.)

* The B$LEV and C$LEV interrupts (RST 6.5 and RST 5.5 pins on the 8085) are level-triggered. A high level occurring on either of these pins while the corresponding interrupt level is unmasked and enabled will cause an interrupt to be acknowledged.

Because of the triggering sensitivity of the B$LEV and C$LEV interrupts, it is the responsibility of the user tasks waiting on the RQLBEX and RQLCEX interrupt exchanges to remove the interrupting signal before a lower-priority task is activated. If this is not done, the same interrupt could be processed repeatedly. The A$LEV interrupt is automatically reset when control is passed to the corresponding interrupt routine or task.

The 8085 on-chip interrupt TRAP is not maskable by any means. Because it is not restricted by any mask or hardware Disable Interrupts (DI) command, it can interrupt any routine, even in "protected regions" of code which should never be interrupted. Therefore its use is strongly discouraged for all but emergency situations (such as a power fail/restart capability). No interrupt exchange is associated with the TRAP level, since it should never cause a call to RQISND (see next paragraph).

An interrupt on the TRAP line is initially vectored by RMX/80 to a HALT command which will stop processing without a context save. A user who wants to use the TRAP interrupt must first make a call to RQSETV for the TRAP level, which will place the address of a special user-supplied TRAP interrupt service routine in the interrupt vector. Such a TRAP interrupt service routine should not make use of any RMX/80 operations; not even RQISND should be called. (RQISND is "protected" from all other interrupts, but cannot be protected from the TRAP interrupt.) As with other user-supplied interrupt service routines, all context saves are the responsibility of the service routine itself.

iSBC 80/10 Interrupts. The iSBC 80/10 has a single hardware interrupt level (level 7), which is transformed via software into an eight-level priority system. RMX/80 then treats these eight levels, in terms of messages and exchanges, as if they were the eight hardware interrupt levels on the iSBC 80/20. The same set of interrupt exchanges (RQL0EX, ...,RQL7EX) is defined, and an interrupt exchange is created and initialized in the same way (during RMX/80's initialization using the configuration module, or at run time via RQCXCH and RQELVL). The sequence of events described under "iSBC 80/20 Interrupts" holds true with two exceptions: the enabling and disabling of interrupts differs, and RMX/80 must consult user-supplied polling routines before processing the interrupt.

On the iSBC 80/10, the rules for enabling and disabling interrupt levels are different. Because interrupt levels 0 through 7 on the 80/10 are actually software pseudo-interrupt levels, all corresponding to the same hardware interrupt, the only two possible conditions are "all interrupts enabled" and "all interrupts disabled." If the running task on the 80/10 has a software priority level in the range 0 through 128, (this will generally be an interrupt-driven task), RMX/80 automatically disables interrupts. If the running task has a software priority in the range 129 through 255, RMX/80 enables all interrupts. The RQELVL and RQDLVL operations have no interrupt enabling or disabling effect on the iSBC 80/10.

When the iSBC 80/10 accepts an interrupt, it disables all subsequent interrupts until the current interrupt has been serviced. Servicing an interrupt on the iSBC 80/10 typically requires an input or output operation; the hardware interrupt remains outstanding and continues to interrupt the system until an I/O operation occurs.

Because all interrupts arrive at the same hardware level, RMX/80 cannot determine which device generated the interrupt, and consequently what software pseudo-interrupt level is concerned. Therefore, you must provide a processing routine (polling routine) for each class of devices that may generate an interrupt. The RQSETP operation is provided for this purpose; further details on this operation are given earlier in this chapter. Each of your polling routines must perform at least the following operations:

- Test the hardware associated with the procedure to determine if it produced the interrupt.
- If no interrupt originated from the hardware managed by the procedure, return the byte value FALSE = 00H (in the accumulator) to the calling program.
- If an interrupt is present:
  a. Process it by acknowledging the hardware as required;
  b. Send an interrupt message to the associated interrupt exchange (IED=RQL0EX, ...,RQL7EX) by calling RQISND, *or* process the interrupt directly; and
  c. Return the byte value TRUE=0FFH (in the accumulator) to the calling program.

These routines must not enable interrupts.

Note that these routines are not interrupt service routines, insofar as they do not need to save context, because RMX/80 does this before calling them. The routines are invoked in order of priority by RMX/80 until one of them returns a TRUE value, or until all eight have been called. In the event that all interrupt procedures are called and none of them indicate that the interrupt was acknowledged, the iSBC 80/10 will enter the halt state with interrupts disabled. This is a catastrophic error condition.

Further note that if your application requires fewer than eight interrupt levels, you need not provide interrupt polling routines for the unused levels; the default interrupt routine for all eight levels is a routine that always returns FALSE.

Because the 80/10 lacks priority interrupt logic, RMX/80 disables all interrupts when it activates a task with priority less than or equal to 128. This makes it particularly important that your application code not disable and enable interrupts.

PL/M programmers must declare the polling routine as a byte-type procedure and return a TRUE (0FFH) or FALSE (00H) indication to RMX/80. Assembly-language programmers must load the accumulator with 0FFH if the routine accepts responsibility for the interrupt or 00H if not, then execute a RET (return) instruction. (Since the subroutine is invoked by a CALL instruction, the return address is on the top of the stack. If the subroutine uses the stack, it must also clear its own data from the stack before returning so that the return functions properly.)

The RQISND operation functions normally in the iSBC 80/10 environment. It acknowledges the interrupt and sends a message to an interrupt exchange. However, you must be certain that the task invoked by the RQISND operation acknowledges the interrupt if the RQSETP routine does not do so.

iSBC 80/10 users can totally replace the RMX/80 interrupt software by directing interrupts to user-supplied routines via an RQSETV operation. If the user directs all level 7 interrupts to user code, RMX/80 does not even know when an interrupt occurs. Of course, the user must assume total responsibility for all interrupts. In this situation the RQSETP operation has no meaning, since RMX/80 does not take part in interrupt processing. RQISND is the only operation with any meaning in such user-supplied interrupt service routines on the iSBC 80/10.

The following PL/M listing illustrates a typical user-supplied interrupt service routine for the iSBC 80/10. This procedure performs the same function on an 80/10 system as the example under "iSBC 80/20 Interrupts" does on an 80/20.

```
SERVICE:
DO;

/* EXTERNALS */

DECLARE DATA$BUFFER(1) BYTE EXTERNAL;
DECLARE DATA$PNTR ADDRESS EXTERNAL;
DECLARE DATA$COUNT ADDRESS EXTERNAL;

RQISND:
    PROCEDURE(IED$ADDRESS) EXTERNAL;
        DECLARE IED$ADDRESS ADDRESS;
    END RQISND;

RQENDI:
    PROCEDURE EXTERNAL;
    END RQENDI;
```

```
/* PUBLICS */

DECLARE RQL7EX(15) BYTE PUBLIC;

INT$SVC:
    PROCEDURE BYTE;

        DECLARE DATA$PORT LITERALLY '0ECH',
                STATUS$PORT LITERALLY '0EDH',
                RECEIVE$READY LITERALLY '002H';

        IF INPUT(STATUS$PORT) AND (RECEIVE$READY > 0) THEN DO;
                DATA$BUFFER(DATA$PNTR) = INPUT(DATA$PORT);
                DATA$PNTR = DATA$PNTR + 1;
                IF DATA$PNTR <= DATA$COUNT THEN CALL RQISND(.RQL7EX);
                RETURN(0FFH);
                END;
        ELSE RETURN(00H);
    END INT$SVC;

END SERVICE;


BLOCK$READ:
DO;

/* PUBLICS */

DECLARE FOREVER LITERALLY 'WHILE 1';
DECLARE DATA$BUFFER(256) BYTE PUBLIC;
DECLARE DATA$PNTR ADDRESS PUBLIC;
DECLARE DATA$COUNT ADDRESS PUBLIC;
DECLARE REQ$EXC EXCHANGE$DESCRIPTOR PUBLIC;

/* EXTERNALS */

DECLARE RQL7EX(15) BYTE EXTERNAL;

INT$SVC:
    PROCEDURE EXTERNAL;
    END INT$SVC;

RQSETP:
    PROCEDURE(SERVICE$ADDRESS,INT$LEVEL) EXTERNAL;
        DECLARE SERVICE$ADDRESS ADDRESS,INT$LEVEL BYTE;
    END RQSETP;

RQSEND:
    PROCEDURE(EXCHANGE$ADDRESS,MESSAGE$ADDRESS) EXTERNAL;
        DECLARE(EXCHANGE$ADDRESS,MESSAGE$ADDRESS) ADDRESS;
    END RQSEND;

RQWAIT:
    PROCEDURE(EXCHANGE$ADDRESS,TIME$LIMIT) EXTERNAL;
        DECLARE(EXCHANGE$ADDRESS,TIME$LIMIT) ADDRESS;
    END RQWAIT;
```

```
USART$READER:  /*MUST BE A TASK WITH PRIORITY 113-128*/
     PROCEDURE PUBLIC:
          DECLARE ENABLE$USART LITERALLY '037H',
                    DISABLE$USART LITERALLY '033H',
                    USART$CONTROL LITERALLY '0EDH';
          DECLARE MSG$PNTR ADDRESS;
          DECLARE MSG BASED MSG$PNTR STRUCTURE(
                                        LINK ADDRESS,
                                        LENGTH ADDRESS,
                                        TYPE BYTE,
                                        HOME$EXCHANGE ADDRESS,
                                        RESPONSE$EXCHANGE ADDRESS,
                                        ADDR ADDRESS);
          DECLARE INTR$MSG ADDRESS;

          CALL RQSETP(.INT$SVC,7);

          DO FOREVER;
               MSG$PNTR = RQWAIT(.REQ$EXC,0);
               DATA$PNTR = 0;
               DATA$COUNT = 256;
               OUTPUT(USART$CONTROL) = ENABLE$USART;
               INTR$MSG = RQWAIT(.RQL7EX,0);
               OUTPUT(USART$CONTROL) = DISABLE$USART;
               MSG.ADDR = .DATA$BUFFER;
               CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSG$PNTR);

          END; /* OF DO FOREVER */

     END USART$READER;

END BLOCK$READ;
```

Details of System Operation

The information given below is intended primarily for debugging purposes and for those who want additional details of system operation. You should not need this information to write tasks for the RMX/80 environment. You should, however, be familiar with this information if you plan to code configuration modules in PL/M.

Figure 2-8 at the end of this chapter is a summary, in diagram form, of the information provided in this section. It is designed to serve as a convenient reference for understanding the system and for debugging.

System Initialization

RMX/80 is designed to initialize a system without the need for any external program loading devices. The system initializes itself by building control structures in RAM from structures supplied by the user in the configuration module. Initialization without external storage devices requires that these configuration structures be located in ROM. In addition, the user must reserve the RAM required for the control structures to be generated.

Three user-supplied structures are required for system initialization. The first of these is the Create Table (RQCRTB). This table is simply a set of pointers to the other two structures, the Initial Task Table and the Initial Exchange Table. The In-

itial Task Table (ITT) is a list of Static Task Descriptors; the Initial Exchange Table (IET) is a list of addresses where the system is to build Exchange Descriptors. Figure 2-3 shows the relationship between the Create Table, the Initial Task Table, and the Initial Exchange Table.

Whenever the system is started, the RMX/80 Nucleus uses these structures in ROM to build Task Descriptors and Exchange Descriptors in RAM. RMX/80 performs an RQCTSK operation for each Static Task Descriptor in the Initial Task Table. (There must always be at least one Static Task Descriptor in the table.) The Nucleus builds an Exchange Descriptor for each exchange address specified in the Initial Exchange Table. Each Exchange Descriptor is built via an RQCXCH operation.

Only the tasks and exchanges specified in the configuration module are created at initialization. However, other tasks and exchanges may be created at run time by application tasks or RMX/80 extension tasks via the RQCTSK and RQCXCH operations.

After the RMX/80 Nucleus builds the Task Descriptors and Exchange Descriptors specified in the configuration module, it determines which task has the highest priority and passes control to that task.

Static Task Descriptor

A diagram of the Static Task Descriptor is given in figure 2-4. The fields of this structure are described below.

NAME contains a string of six ASCII characters giving the name of the task.

INITIAL PROGRAM COUNTER contains the address of the first instruction to be executed by the task.

STACK ADDRESS contains the (lowest numeric) address of the RAM to be used as the task's stack.

STACK LENGTH contains the number of bytes of RAM allocated to the task's stack.

PRIORITY contains the task's software priority level.

INITIAL EXCHANGE contains the address of the task's initial default exchange, or zero if there is no default exchange.

TASK POINTER contains the starting address of the 20 bytes of RAM to be used as the task's Task Descriptor.
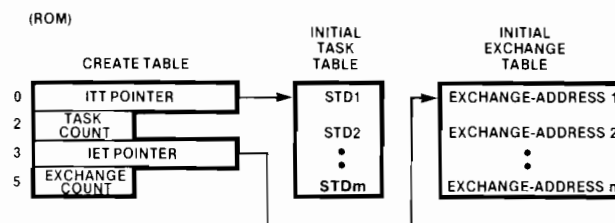
Figure 2-3. System Initialization Structures

Task Descriptor

Structure. Some of the information in the Task Descriptor is simply transferred directly from the Static Task Descriptor. The Task Descriptor is built at the RAM address specified by the TASK POINTER field in the Static Task Descriptor, as shown in figure 2-5. Arrows show values moved from the Static Task Descriptor to the Task Descriptor when the system is started.

The STACK POINTER in the Task Descriptor is set equal to STACK ADDRESS + STACK LENGTH from the Static Task Descriptor. The MARKER field is the lowest numeric address available for the task's stack. The Debugger uses the MARKER field to detect stack overflow. The RMX/80 Nucleus does not check for stack overflow because of the overhead this would impose on the system.

After the task's stack is created, RMX/80 pushes the INITIAL PROGRAM COUNTER from the Static Task Descriptor onto the stack. Whenever a task is not running, its program counter is stored on the stack. RMX/80 fills the remainder of the stack with the value 0C7H. This is an additional debugging aid, since it makes it possible for you to see how much of the stack has been used.

Notice that the EXCHANGE ADDRESS field is initially set to the corresponding value from the Static Task Descriptor. This field provides a default exchange address for RQWAIT and RQACPT operations that specify an exchange address value of zero. However, this address is altered when the task is running. EXCHANGE ADDRESS always points to the exchange where the task is currently waiting or where it last waited or accepted a message; the value in this field is reset each time the task names an exchange in an RQWAIT or RQACPT operation.

DELAY LINK FORWARD and DELAY LINK BACK are used for entering the Task Descriptor on the system Delay List when necessary. these point to the next and previous tasks on the Delay List.
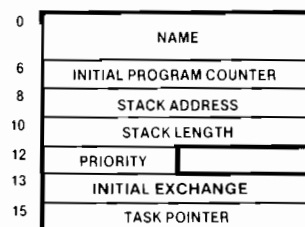
Figure 2-4. Static Task Descriptor
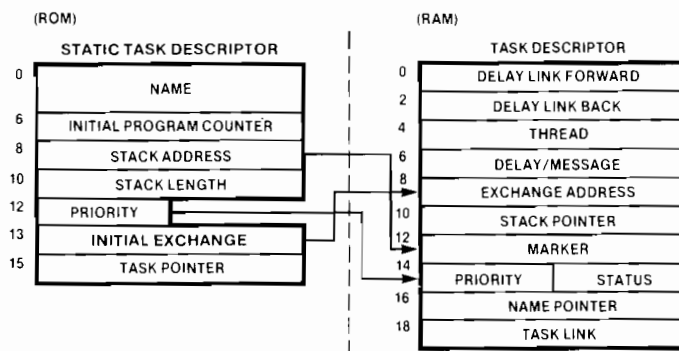
Figure 2-5. Static Task Descriptor - Task Descriptor Relationships

THREAD is a general-purpose field used to link the Task Descriptor onto the Ready List or Suspend List. This field is also used to link the Task Descriptor onto an exchange's list of waiting tasks.

DELAY/MESSAGE is another multi-purpose field. When the task is in a timed wait, this field of the first Task Descriptor on the Delay List indicates the remaining delay time, expressed in system time units. The first task on the Delay List has the least remaining delay. In subsequent Task Descriptors on the Delay List, the field indicates the additional delay required for that task after the delay for the preceding task has expired. When the task receives a message, the address of the message is temporarily stored in this field. When the task becomes the running task, this address is placed in the H and L registers.

STATUS is a one-byte field. Individual bits of this field indicate task status:

- Bit 0, DELAYED$TASK: Bit 0 is set to 1 if the task is on the Delay List.

- Bit 1, SUSPENDED$TASK: Bit 1 is set to 1 if the task is on the Suspend List or if it is to be placed there when it next becomes ready.

- Bit 2, PREEMPTED$TASK: Bit 2 is set to 1 when the task is preempted by a hardware interrupt.

- Bits 6 and 7, DEBUG$BITS: Bits 6 and 7 are reserved for use by the RMX/80 Debugger.

NAME POINTER contains the address of the Static Task Descriptor used to create the task (and thus the address of the task's name).

TASK LINK links the Task Descriptor to a list of all the tasks in the system.

Accessing the Task Descriptor. A task must not alter its own or any other task's Task Descriptor. However, it may be useful for a task to read its own EXCHANGE ADDRESS, NAME POINTER, PRIORITY, and MARKER fields. If a task does not know the address of its own Task Descriptor, the Task Descriptor can be accessed through the public address variable RQACTV. RQACTV always points to the Task Descriptor of the running task.

Exchange Descriptor

Structure. Exchange Descriptors have the format shown in figure 2-6.


```
0 |  MESSAGE HEAD  |
2 |  MESSAGE TAIL  |
4 |   TASK HEAD    |
6 |   TASK TAIL    |
8 | EXCHANGE LINK  |
```
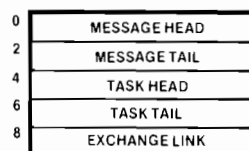
Figure 2-6. Exchange Descriptor

MESSAGE HEAD is a pointer to the first message waiting at the exchange. This field is set to zero if no message is waiting at the exchange.

MESSAGE TAIL is a pointer to the last message waiting at the exchange. If no message is waiting, this field is set to the address of the exchange.

TASK HEAD is a pointer to the first task waiting at the exchange. If no task is waiting, this field is set to zero.

TASK TAIL is a pointer to the last task waiting at the exchange. If no task is waiting, this field is set to the address of the exchange.

EXCHANGE LINK contains the address of the next Exchange Descriptor in the list of all the Exchange Descriptors in the system.

Note that because of the way tasks and messages are queued at exchanges, a task can wait at only one exchange at a time.

Use in System. The following example illustrates the use of an exchange. For the sake of simplicity, assume that the Exchange Descriptor is located at address 4500; the messages sent to the exchange are at locations 4100, 4200, and 4300. When first created, the Exchange Descriptor is set to the following values (the use of the EXCHANGE LINK field is not shown):

| 0 |
| 4500 |
| 0 |
| 4500 |
| xx |

Assume that a task sends the message at location 4100 to the exchange. The MESSAGE HEAD and MESSAGE TAIL fields change as shown:

| 4100 |
| 4100 |
| 0 |
| 4500 |
| xx |

Since there is only one message at the exchange, the MESSAGE HEAD and MESSAGE TAIL fields are the same. TASK HEAD and TASK TAIL remain unchanged.

Now assume that a task sends the message at location 4200 to the exchange. The Exchange Descriptor is changed as shown:

| 4100 |
| 4200 |
| 0 |
| 4500 |
| xx |

When the third message is sent, the Exchange Descriptor is altered as follows:

| 4100 |
| 4300 |
| 0 |
| 4500 |
| xx |

Notice that address 4200 no longer appears in the Exchange Descriptor. This raises the question of how the system knows where the second message is located. This information is stored in the LINK field of the message heading. The LINK field (the first two bytes) of the first message contains the address of the second message; the LINK field of the second message contains the address of the third message.

Now consider the effect of a task issuing an RQWAIT at this exchange. There is no need to queue the task at the exchange, since there is a message available. (Tasks are queued at exchanges using the same techniques as messages. When multiple tasks wait at the exchange, they are linked together by entries in the THREAD field of the Task Descriptor. In this case, the THREAD field serves the same purpose as the LINK field of the message heading.)

The waiting task is given access to the first message waiting at the exchange by moving the address in MESSAGE HEAD to the DELAY/MESSAGE field in the Task Descriptor. RMX/80 updates the MESSAGE HEAD by moving the address in the LINK field of the first message into MESSAGE HEAD. The Exchange Descriptor is updated so that the message at location 4200 is the first message waiting at the exchange.

Interrupt Exchange Descriptor

Structure. As shown in figure 2-7, an Interrupt Exchange Descriptor is fifteen bytes long and consists of two parts. The first ten bytes correspond to the ten bytes of a regular Exchange Descriptor, and the fields are used in exactly the same way. The last five bytes are a short message consisting only of the three required message fields (LINK, LENGTH, and TYPE).

The two parts of the Interrupt Exchange Descriptor are logically separate; they are contiguous in memory only for convenience to the RMX/80 software. An interrupt message need never be longer than five bytes, since the only information it needs to convey is that an interrupt has occurred at the corresponding level, and whether or not there has been a missed interrupt.

Use in System. As discussed under "iSBC 80/20 Interrupts," one Interrupt Exchange Descriptor must be defined for each interrupt level that is used in the system. The Interrupt Exchange Descriptor is initialized either at system initialization time using information in the configuration module, or at run time via an RQCXCH operation (for the Exchange Descriptor portion) and an RQELVL operation (for the message portion). When an interrupt occurs at a given level, RMX/80 sends the message portion of the appropriate Interrupt Exchange Descriptor to that interrupt exchange—i.e., it takes the address of the message portion and adds it to the list of messages linked to the Interrupt Exchange Descriptor. The TYPE field of the message is set to INTMSGTYPE(1). Before it sends the interrupt message to the
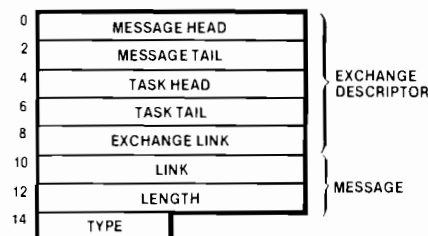
Figure 2-7. Interrupt Exchange Descriptor

exchange, RMX/80 checks to see if there is already an interrupt message queued at the exchange. If so, the queued message's TYPE is set to MISSEDINTTYPE(2) and the second interrupt message is simply dropped.

The following example illustrates the use of an interrupt exchange. For simplicity, assume that the Interrupt Exchange Descriptor is located at address 5000.

When first created, the Interrupt Exchange Descriptor is set to the following values (the use of the EXCHANGE LINK field is not shown). Note that the LINK field of the interrupt message is set to its own address. This indicates that it is not queued at the exchange.

0	0
2	5000
4	0
6	5000
8	XX
10	5010
12	5
14	1

Assume that an interrupt causes an interrupt message to be sent to this exchange. The MESSAGE HEAD and MESSAGE TAIL pointers are set to the address of the interrupt message, and the message's LINK field is set to zero. The Interrupt Exchange Descriptor will therefore appear as follows:

0	5010
2	5010
4	0
6	5000
8	XX
10	0
12	5
14	1

If another interrupt message is posted at the exchange before a task has become available to receive the first message, the IED will be changed as shown below. (The type of the interrupt message will be changed to MISSEDINTTYPE(2), but the new message will not be queued at the exchange.)

0	5010
2	5010
4	0
6	5000
8	XX
10	0
12	5
14	2

Now suppose that a task arrives to wait at the exchange. The task will receive the first message; i.e., the address in the MESSAGE HEAD field of the Interrupt Exchange Descriptor will be moved to the DELAY/MESSAGE field in the Task Descriptor. The address in the LINK field of the message—in this case zero, since only the first message is queued at the exchange—is moved into MESSAGE HEAD. MESSAGE HEAD is now zero, indicating that no messages are left in the queue, and MESSAGE TAIL is set to point to the address of the exchange.

0	0
2	5000
4	0
6	5000
8	XX
10	0
12	5
14	2

The receiving task (or some other task) can check the TYPE field of the message and determine that one or more interrupt messages were missed.
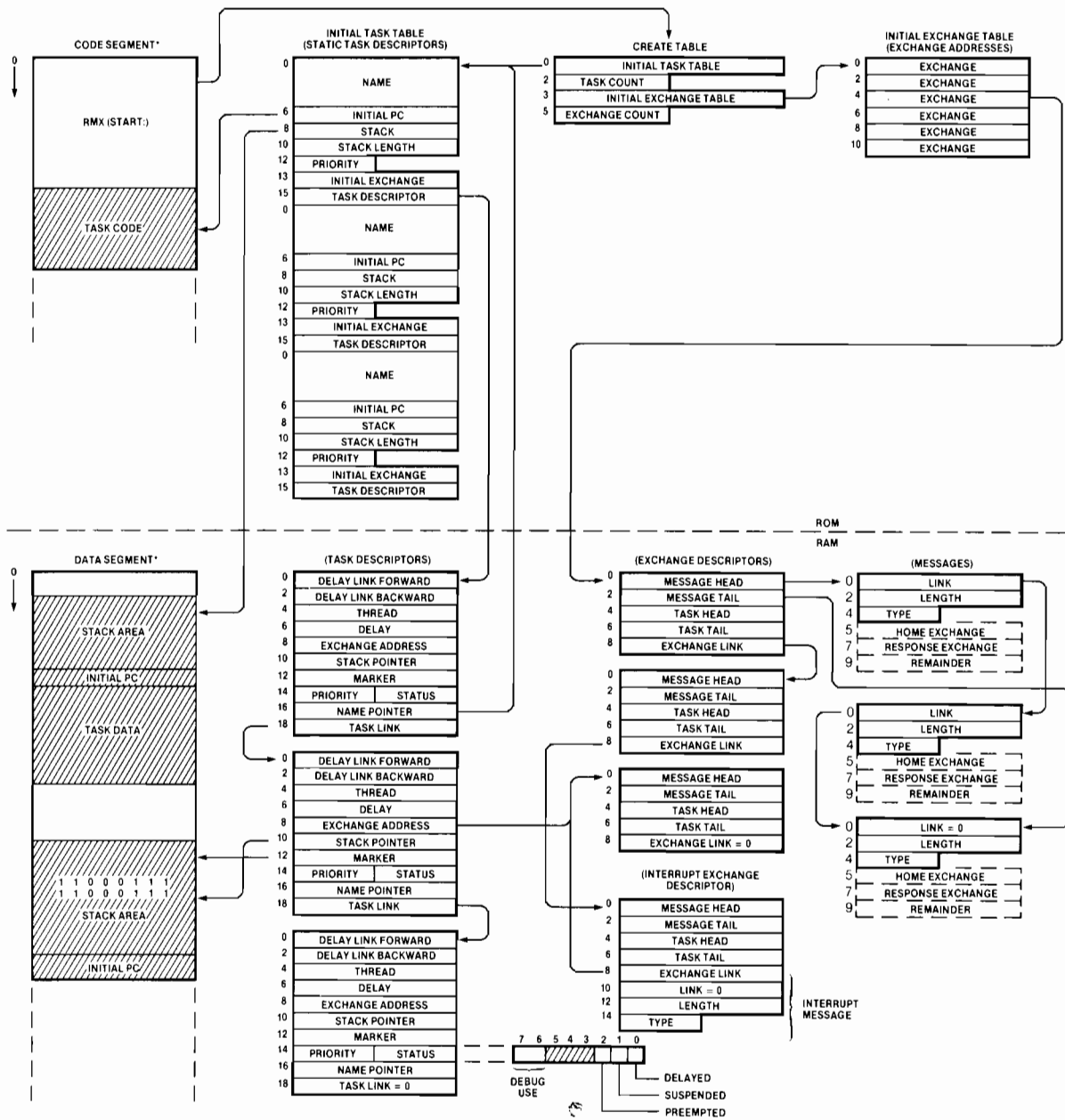
Figure 2-8. RMX/80 System Control Structures

This chapter provides instructions for building your application system around RMX/80 software.

Although developing a real-time, multitasking system is a creative process, a discussion of the steps involved can significantly reduce the amount of time expended during system development. Such a discussion is provided within this chapter.

Additionally, this chapter enumerates the requirements of the Nucleus that must be considered while the system is being developed. It also supplies suggestions for making the most effective use of the features of RMX/80.

In building your RMX/80 application, you must perform the following basic steps:

1. Design the system, defining hardware and software components.
2. Code user tasks and translate them via an Intel language translator.
3. Generate the software configuration — i.e., prepare the configuration module, then link and locate object modules.
4. Test and debug the system.
5. Burn the code into PROM.

Designing Your System

System design consists of determining the answers to these questions:

- What operations must the system perform?
- What hardware components (iSBC products, peripherals) are needed?
- What software components (RMX/80 Nucleus code and extension tasks, user tasks) are needed?
- In what physical memory spaces (ROM, RAM, on-board memory, off-board memory) will the software components and data reside?

The answer to the first question, of course, largely determines the answers to the other three. The other three questions, however, cannot be answered sequentially; hardware, software, and the location of software and data in hardware are closely interrelated. As the design and implementation proceed, changes made in the planned hardware configuration may affect the software configuration, and vice versa. Designing a system should therefore be treated as an iterative process.

During the system development process, the hardware and software configuration usually will, out of necessity, go through a series of changes. It will generally be desirable to test some hardware and software components before others are ready. The Intellec Microcomputer Development System, with its resident diskette operating system (ISIS-II) and the ICE-80 or ICE-85 In-Circuit Emulator option, facilitates the development of an evolving system. ICE-80 and ICE-85 include some debugging features, and also permit the use of Intellec memory in place of part or all of the iSBC 80 physical RAM and/or ROM (memory mapping facility). The RMX/80 software package also includes a Debugger, described in Chapter 6, which is especially designed for real-time systems based on RMX/80. You may wish to configure the Debugger in your system during development, then delete it when the rest of the system is fully tested.

Hardware Components

Design of a microcomputer system using iSBC hardware is outside the scope of this manual. It is sufficient here to make a few general comments, and to mention some constraints and considerations related to accommodating hardware design to RMX/80 software.

Requirements. Certain special installation procedures must be followed when using an Intel terminal, ICE-80 or ICE-85, and/or the RMX/80 Demonstration System program on an iSBC system. These procedures are given in Appendix F, "Hardware Considerations."

The RMX/80 Nucleus has certain hardware requirements, which are listed in Chapter 2. Hardware requirements for RMX/80 extensions are given in the chapters on those extensions. You also need to estimate the I/O, interrupt, and memory requirements for your application tasks. Refer to "Planning Memory Space" later in this section for further discussion of memory requirements.

Software Components

Your RMX/80 application will consist of the RMX/80 Nucleus (always required) and two main categories of tasks — those selected from the RMX/80 software package and those supplied by you, the user, for your particular needs.

RMX/80 Extension Tasks. You will probably want to select these first. Most RMX/80 extensions provide several tasks, from which you may choose those you need for your application. RMX/80 extensions are covered in later chapters of this manual (Chapter 4 and all the following chapters); refer to these chapters for functional descriptions of the tasks. Appendix D provides a complete list of RMX/80 extension tasks and the memory requirements for each.

User Tasks. Breaking down your application into tasks is an extremely important part of the system design process, and should be done with care. In general, it is better to have a number of smaller tasks rather than a few very large tasks, in order to minimize system idle time. However, the number of tasks should not be too great, either. Too many tasks add needless complexity to the system, making it difficult to modify or debug, and also add to system overhead. In general, tasks should be allocated to performing *functions* and servicing *events*.

Another general rule to follow is that high-priority tasks, including those that service interrupts, should be as short as possible. To understand the reason for this rule, consider a task that performs a number of lengthy calculations and is invoked by a high-priority interrupt. As long as this task is running, no lower-priority interrupt can be serviced, and no lower-priority task can be run. The case of lower-priority interrupts is especially troublesome, since it is possible that some interrupts might be missed altogether. A reasonable solution to this problem is to divide the task into two smaller tasks. The first task services the interrupt as quickly as possible and then sends the input data (in the form of a message) to a second task. The second task, which may have a much lower priority, can then process the data, allowing other interrupts to be serviced and other tasks to compete for the processor.

Exchanges. After defining your tasks, you must determine the exchanges you need for intertask communication. To the exchanges required for the RMX/80 extensions, you add those you define yourself for the needs of your own tasks. There is generally some freedom of choice in defining the number of exchanges; some tasks may be able to share an exchange, and others may operate more efficiently with dedicated exchanges. Later, when you test your system, you may want to alter your exchange configuration to improve system performance.

Once you have selected your tasks and exchanges, it is helpful to make a system diagram showing task-exchange relationships. (For examples, see the exchange diagrams near the beginning of Chapters 4 through 8, and also figures J-2 and J-3 in Appendix J.)

Planning Memory Space

As shown in figure 2-8 at the end of Chapter 2, several types of code and data segments must be allocated memory space in an RMX/80 application system. These segments may be summarized as follows:

ROM segment:

- RMX/80 Nucleus code
- Initialization structures required by the Nucleus (Create Table, Initial Task Table, and Initial Exchange Table)
- Code for RMX/80 extension tasks
- Code for user application tasks

RAM segment:

- Data area required by RMX/80 Nucleus (for list pointers, system messages, etc.)
- Task Descriptors and Exchange Descriptors
- Stack, initial program counter, and miscellaneous data required by user application tasks in the system (including messages)

As explained in the footnote on figure 2-8, these sections of code and data need not be located in the particular address spaces or sequence shown in the diagram. The only constraints are that the RMX/80 START module (which is the main program module in any RMX/80 application system) be located at address 0H, or address 40H or higher. In the later case, RESET will not restart the system unless user code at location 0H transfers control to the Nucleus START module. The first section of RAM must be located at the address determined by the hardware configuration (refer to "Linking and Locating" later in this chapter.)

It is important to remember that when an RMX/80 system is set up to be self-initializing without the use of peripheral loading devices, all information in the initial configuration must be in ROM. This means that any constants to be stored initially in RAM must be provided in ROM, along with the code to store them in RAM after the program is started. PL/M INITIAL declarations and initial values specified in assembler DSEG directives have no meaning in such a system unless RAM is loaded from an external device.

Appendix D lists the memory requirements in bytes for RMX/80 Nucleus and extension task code and data, and for system structures in ROM and RAM. Note that future enhancements to RMX/80 may result in changes to the exact numbers of bytes required.

Some RMX/80 extensions impose special memory requirements. For example, the Disk File System tasks require a certain amount of controller-addressable RAM, which for iSBC 80/20 and 80/10 systems must be located off the main CPU board (so that it is accessible to the bus). Refer to the chapters on the individual extensions you are using for any special requirements.

Coding User Tasks

General System Structure

An RMX/80 application system consists of a main program module (the RMX/80 START module) and an arbitrary number of independent tasks that run under the control of the main module. The main module and any of the tasks may call pro-

cedures or subroutines in a nested fashion, subject to the constraints of the programming language and RAM space available for stacks. User-supplied application tasks and RMX/80 extension tasks are treated identically by RMX/80.

It is important to recognize the difference between a task and a procedure (PL/M) or subroutine (assembly language). A task is scheduled for execution by RMX/80 and may run concurrently with other tasks. A procedure or subroutine is called by the RMX/80 main program module, by a task, or by another procedure or subroutine, and gains control immediately when it is called. A task is invoked by the RMX/80 scheduling software; a procedure or subroutine is invoked by an 8080 or 8085 CALL instruction.

Another crucial difference between a task and a procedure or subroutine is that a procedure or subroutine may be thought of as a section of code, whereas a task cannot always be visualized this way. A task is defined by its Static Task Descriptor, which gives a starting address for code. It is possible for several tasks to run with the same section of code, if that code is made reentrant. (See "Code Shared by More Than One Task" later in this section.)

Unlike a procedure or subroutine, each RMX/80 task requires a separate stack for context saves; there is no system stack. Your task may also use its stack for its own processing. RMX/80 performs all context saves for tasks. However, if you wish to supply your own interrupt service routines (as described under "Interrupt Handling" in Chapter 2), these routines must perform a context save on entry and a context restore on exit.

Parameter Passing and Returned Values

All RMX/80 operations follow the PL/M parameter passing conventions. For this reason, and also because you may wish to code some of your application software in PL/M and some in assembly language, it is well to note the parameter passing conventions followed by PL/M. If there is one parameter, it is passed in the B and C registers if it is an address value, or in the C register if it is a byte value. If there are two parameters, the first is passed in the B and C register pair. The second parameter is passed in the D and E registers if it is an address value, or the E register if it is a byte value. If there are more than two parameters, the first n-2 are passed on the stack (earlier parameters are pushed first) and the last two are passed in the BC and DE register pairs.

If a procedure returns an address value, it is returned in the H and L register pair. If the procedure returns a byte value, it is returned in the accumulator.

Public and External Symbols

To establish linkage among tasks and RMX/80 software, the modules that contain code for the tasks must declare certain public and external symbols. For example, the address of Exchange Descriptors must be declared PUBLIC by the program module where they are defined and EXTERNAL (EXTRN in assembly language) by any other modules that reference them. Appendix B supplies a list of all RMX/80-defined PUBLIC and EXTERNAL symbols with which your tasks may need to interface.

Names

Names of variables and procedures or subroutines in user tasks may be any names permitted by the programming language, except that no names should have a "Q" or "?" in the second position. Names beginning with "RQ" are reserved for RMX/80 identifiers.

Programming Aids

The RMX/80 diskettes provide a number of "INCLUDE files" — i.e., files containing PL/M or assembly-language code commonly needed in user tasks. Rather than write out the code yourself wherever you need it in your program, you can simply name the appropriate file in an assembler or PL/M compiler $INCLUDE control, and the code will be inserted in your program. Appendix A lists the names of all INCLUDE files on the diskettes and indicates the contents of each. To make sure the names you reference elsewhere in your tasks match those in the INCLUDE files you use, it is advisable for you to make listings of these files ahead of time.

Initialization

Tasks are typically coded as an endless loop preceded by a segment of initialization code. The initialization code is usually specific to the individual task and includes functions such as initializing counters. Instead of having each task perform its own initialization, you may prefer to code one task to perform the initialization for all other tasks. However, this scheme may involve the declaration of a large number of public symbols, which may prove cumbersome, especially when several programmers are coding individual tasks for the system.

Code Shared by More than One Task

As discussed earlier, it is possible for several tasks to share the same section of code, if that code is made reentrant. (Reentrant code keeps all variable data in registers and/or on the stack. See the PL/M-80 Programming Manual for details.) Information (such as display messages) that differs from task to task can be supplied in a message sent to invoke the task.

To accomplish this, you specify (in the configuration module or via RQCTSK operations) a different Static Task Descriptor for each task, but assign the same Initial Program Counter value in each Static Task Descriptor. (This cannot be done if you use the assembly language STD macro.) If you are creating the tasks dynamically, an alternative method is to use a single Static Task Descriptor in RAM and modify the TASK POINTER and STACK ADDRESS fields of the Static Task Descriptor before each call to RQCTSK.

In addition, each of the tasks sharing the same code may need to wait for a message at its own unique exchange. The shared code can accomplish this by using the default exchange feature of the RQWAIT or RQACPT operation. If RQWAIT or RQACPT is called with an exchange address of zero, the task waits at the default exchange specified in the EXCHANGE ADDRESS field of the Task Descriptor.

There is, however, one set of circumstances under which this technique for providing unique exchanges fails. Suppose that three tasks (called A, B and C) share a code segment. Further suppose that this common code segment causes each task to wait at two exchanges, and that the first exchange at which each task waits is intended to be distinct from the exchanges at which the other two tasks wait. In other words, task A waits at exchange 1, task B at exchange 2, and task C at exchange 3. Also suppose that the second exchange at which each task waits (exchange 4) is common to all three tasks. Using the default exchange technique described in the preceding paragraph, each task will initially wait at the proper exchange. However, once a task waits at exchange 4, the default exchange address associated with that task is modified by RMX/80 to point to this most recently used exchange. Consequently, each task loses access to its unique exchange.

The default exchange technique can still be used if the following precaution is taken. Before waiting at the common exchange, the shared code segment must extract the address of the default (unique) exchange from the Task Descriptor of the running task (referenced by RQACTV). This address should then be stored in local storage. After the shared code segment has caused the task to wait at the common exchange,

the address of the unique exchange should be retrieved from local storage and stored back into the Task Descriptor. This process, although cumbersome, will preserve the uniqueness of unshared exchanges.

Messages

RMX/80 imposes certain format requirements on all messages in the application system. These are described in Chapter 2 under "Message Coding." More specific requirements for particular types of messages sent to RMX/80 extension tasks are given in the chapters on the RMX/80 extensions.

When a task sends a message via an RQSEND operation, what is sent is the address of the first byte of the message. In PL/M, you may use this address as a base and treat the message as a based variable. If you are programming in assembly language, you will find this address in the H and L registers, which can be incremented to access the other fields of the message.

Once it is sent, a message is intended to be the exclusive property of the task that receives it; neither the sending task nor any other task should alter or resend it. After the receiving task is finished with the message, it may send it back to the originating task or to another task. By convention, this is done using the RESPONSE EX-CHANGE or HOME EXCHANGE fields of the message.

Interrupts

Application tasks should not directly execute 8080 EI and DI instructions. If the task needs to disable a particular interrupt on the iSBC 80/20 or 80/30, RQDLVL should be used; RQELVL should be used to re-enable it. In the iSBC 80/10 version of RMX/80, the priority of the running task determines whether the single hardware interrupt level is enabled or disabled. (Refer to "iSBC 80/10 Interrupts" in Chapter 2.)

Exclusive Access to Code

When an application task requires exclusive access to a section of code, locking out even tasks of higher priority that need to use it, a software lock can be programmed using the RQSEND and RQWAIT operations.

For example, assume that a system contains a non-reentrant software divide routine. Because this can be a lengthy routine, only one copy of it exists in the system, and it is shared by all tasks. Because the RAM buffers must not be altered by a second user during the division, it becomes necessary to block access to the division routine. The routine may be preempted and restarted, but no other user can be allowed to use the routine. The lock can be implemented as follows:

```
/*TASK INITIALIZATION CODE*/
    .
    .
    .
CALL RQSEND(.LOCKEX,.LOCK);
/*END OF INITIALIZATION CODE*/

/*DIVIDE ROUTINE*/
GO=RQWAIT(.LOCKEX,0);
    .
/*INSERT DIVISION ROUTINE CODING HERE*/        }  protected code
    .
    .
CALL RQSEND(.LOCKEX,GO);
```

The first task that attempts to use the divide routine will be able to do so, since there is a single message waiting at the LOCKEX exchange as a result of the task initialization code. However, as soon as this first task begins executing the division routine, it also removes the message from the LOCKEX exchange. This ensures that the task retains control of the division routine and thus protects its RAM buffers until the division is complete. At this point, the LOCK message is sent back to the LOCKEX exchange to enable another task to obtain access to the routine.

Generating The Software Configuration

After defining and coding your own tasks, you specify the configuration of your system by means of a section of code known as the *configuration module.* You then link your configuration module to the RMX/80 tasks and user tasks you have selected.

Generating the software configuration for your application system consists of four major activities:

1. Defining system components

2. Preparing the configuration module

3. Preparing the controller-addressable memory module (when the Disk File System is used)

4. Linking and locating object modules.

Items 1, 2, and 4 are discussed in this section; item 3 is covered in Chapter 7.

Throughout this section a single application is used as an example. The key characteristics of this application are:

*   RMX/80 tasks: input-output Terminal Handler, Free Space Manger, and active Debugger

*   User tasks: three, called USER1, USER2, and USER3. USER1 services interrupts from a high-speed peripheral device; USER2 and USER3 are not interrupt-driven.

Defining System Components

Before the configuration module can be coded or link commands can be prepared, several questions must be answered:

*   What tasks are to be included in the system, and what are the characteristics of these tasks (priorities, etc.)?

*   What exchanges are to be defined when the system is initialized?

*   What PUBLIC data items (if any) need to be defined for use by RMX/80 extension tasks?

*   What are the characteristics of each disk controller in the system?

*   What are the characteristics of each disk drive in the system?

*   How are Disk File System buffers to be allocated?

To provide a convenient and uniform framework for answering these questions, a "System Definition Worksheet" is furnished here. Once it is filled out, the worksheet contains all the information needed for coding the configuration module.

The first page of the worksheet (Part I) covers tasks, exchanges, and PUBLIC variables. The second page (Part II) defines controllers, drives, and buffers — the items particular to the Disk File System. For systems that do not include DFS, only Part I (figure 3-1) is required.

The methodology of this section is to explain each item on Part I of the worksheet, and to provide sample entries using the application described at the beginning of the section. Part II of the worksheet is illustrated and covered in Chapter 7.

Tasks. The tasks to be entered on the worksheet are those you defined in your system design, as described earlier in this chapter under "Software Components." Only tasks needed at initialization time need to be entered, not tasks created at run time. Most RMX/80 extension tasks create other tasks at run time; but only those tasks listed in the "Configuration Requirements" section of the appropriate RMX/80 extension chapter are to be entered.

Task Worksheet Entries. As shown in figure 3-1, five items must be entered on the worksheet for each task in the application: name, initial program counter, stack length, priority, and default exchange.

1. TASKS:

2. INITIAL EXCHANGES:

NAME	INITIAL P.C.	STACK LENGTH	PRIORITY	DEFAULT EXCHANGE	NAME

3. PUBLIC DATA:

SYMBOL	VALUE		SYMBOL	VALUE	SYMBOL	VALUE
_____	_____		_____	_____	_____	_____
_____	_____		_____	_____	_____	_____
_____	_____		_____	_____	_____	_____

Figure 3-1. RMX/80 System Definition Worksheet, Part I

Each task must be given a NAME consisting of any six ASCII characters. For RMX/80 extension and disk controller tasks, use the names furnished in the appropriate RMX/80 extension chapters. User tasks may be given any convenient names, except that these names should not have a "Q" or "?" as the second character.

The INITIAL PROGRAM COUNTER for a task is its entry point address, generally given as a symbolic value.

The maximum STACK LENGTH for a task is determined by several factors, which are discussed further below under "Estimating Stack Length."

You must assign a software PRIORITY level between 1 and 254 to each user and RMX/80 extension task. Rules and suggestions for assigning priorities are given below under "Assigning Priority Levels."

The DEFAULT EXCHANGE is optional for user tasks. RMX/80 places this data in the EXCHANGE ADDRESS field of the Task Descriptor it builds when the system is initialized. The default exchange must be supplied for Disk File System service and controller tasks. (Refer to Chapter 7.)

You must determine the worksheet values for all user tasks. The User's Guide chapters on RMX/80 extensions provide the information needed for extension tasks.

Estimating Stack Length. The maximum stack length for a task is a function of the level of nesting which occurs within the task. It also must include space for any miscellaneous data the user task pushes on the stack and 24 extra bytes for use by the RMX/80 scheduling algorithms. A rough estimate of stack length in bytes may be obtained by the following formula:

$$S_T = 24 + 1.10\,(S_U),$$

where S_T is the approximate stack length required for the task; S_U is the compiler- or assembler-generated stack requirement for the user task, including the user-supplied procedures or subroutines and the RMX/80 procedures that it calls. The 24 extra bytes are required for RMX/80 interrupt handling, and the 1.10 multiplier serves as an error factor.

Stack lengths required for RMX/80 extension tasks and Nucleus operations are supplied in Appendix D.

Another approach for determining task stack length is to start with a grossly overestimated value (considerably larger than the value found by the formula above), and then to use the RMX/80 Debugger FORMAT TASK command (see Chapter 6) to determine the amount of stack space left after testing it under worst-case conditions. After subtracting 24 from this value to account for the fact that the task may not have been interrupted, the stack allocation can be correspondingly reduced.

Assigning Priority Levels. RMX/80 provides 256 software priority levels; of these, 254 are available to the user. (Level 0 is used by the Debugger scan and exchange breakpoint facilities; level 255 is reserved for the idle task.)

Priority levels 1 through 128 correspond to specific interrupt levels, as shown in table 2-2 in Chapter 2. These priorities should generally be reserved for interrupt service tasks, with the devices of highest speed running at the highest interrupt levels (lowest level numbers) with service tasks given corresponding software priorities. Tasks that do not service interrupts should generally be assigned priorities 129

through 254. An exception to this may occur when background calculations for a high priority device are more important than handling interrupts from a very low priority device. In this case the background task should have a priority that is higher than the interrupt task for the low priority device.

In general, emergency tasks should have higher (lower-numbered) priorities than routine operations; quickly finished operations should have higher priorities than longer operations; and tasks requiring rapid response should have higher priorities than those that are not so urgent. During testing of the system, it may become apparent that certain tasks are running too frequently or not frequently enough. In such cases, reconfiguring the system with different priorities for these tasks may correct the problem.

Initial Exchanges. Any additional user or extension task exchanges that are to be created at system initialization time should be listed on the worksheet. (Exchanges created at run time via RQCXCH operations should not be included.)

PUBLIC Data. Some RMX/80 extensions require certain data items to be declared PUBLIC and initialized; for instance, the Terminal Handler can require a value for the variable RQRATE. (The chapters that cover the extensions specify any PUBLICs that must be defined.) The most appropriate place to define these data items is in the configuration module. PUBLIC data to be defined can be entered in section 3 of the worksheet.

Worksheet Entries for Example Application. Figure 3-2 shows how the System Definition Worksheet might be completed for our example application system. We have assigned USER1 a priority of 35, which corresponds to interrupt level 2; all other tasks are assigned priorities lower (higher in number) than 128. USER1 requires the RQL2EX exchange, and three other user exchanges (USXCH1, USXCH2, and USXCH3). The task and exchange entries for the Terminal Handler, Free Space Manager, and Debugger are explained in the chapters on these RMX/80 extensions.

Since the Terminal Handler is included (see Chapter 4), the variable RQRATE must be declared PUBLIC and assigned a value. (Note that the 80/20 and 80/30 versions of the Terminal Handler are capable, with help from the operator, of determining the correct value for RQRATE. See Chapter 4 for details.) Assuming that our terminal has a transmission rate or 1200 baud, and that our system CPU board is an iSBC 80/20, the value of RQRATE (per table 4-1) must be 56.

Preparing the Configuration Module

General Instructions. The configuration module may be coded in either PL/M or assembly language. The RMX/80 product diskettes provide assembly language macros for use in system configuration. You will probably find it easier to code your configuration module in assembly language, with the aid of these macros, than to use PL/M.

An RMX/80 configuration module consists of declarations for the following items:

1. Task Stacks (RAM)
2. Task Descriptors (RAM)
3. Task External Declarations
4. Initial Task Table (ROM)
5. Exchange Descriptors (RAM)
6. Interrupt Exchange Descriptors (RAM)
7. Initial Exchange Table (ROM)
8. Create Table (ROM)
9. PUBLIC data items (ROM) if any are required by RMX/80 extension tasks.

(Up to four additional items, discussed in Chapter 7, are required in configuration modules for system using the Disk File System.)

All of the segments listed above are coded from information on the RMX/80 System Definition Worksheet.

The four assembly-language configuration macros provided on the RMX/80 Executive diskettes are described in the following section. Following the descriptions of the macros are listings of PL/M and assembly language configuration modules for our example application system.

RMX/80 SYSTEM DEFINITION WORKSHEET
(PART I)

1. TASKS:

2. INITIAL EXCHANGES:

NAME	INITIAL P.C.	STACK LENGTH	PRIORITY	DEFAULT EXCHANGE	NAME
RQTHDI	RQTHDI	36	112	RQOUTX	RQINPX
RQFMGR	RQFMGR	40	130	RQFSRX	RQOUTX
RQADBG	RQADBG	64	140	RQWAKE	RQWAKE
USER 1	USER 1	24	35	RQL2EX	RQDBUG
USER 2	USER 2	28	135	USXCH2	RQALRM
USER 3	USER 3	48	150	USXCH3	RQL6EX
					RQL7EX
					RQFSRX
					RQFSAX
					RQL2EX
					USXCHI
					USXCH2
					USXCH3
					RESP$EX

3. PUBLIC DATA:

SYMBOL	VALUE	SYMBOL	VALUE	SYMBOL	VALUE
RQRATE	56				

Figure 3-2. Completed System Definition Worksheet for Example Application

Assembly Language Macros. The RMX/80 Executive diskettes for the iSBC 80/20, 80/30, and 80/10 supply a set of assembly language macros to help define the Create Table, the Static Task Descriptors that comprise the Initial Task Table, and the Initial Exchange Table. Using these macros is substantially easier than manually coding definitions in either PL/M or assembly language. The definitions for these macros can be added to a program module by using the assembler $INCLUDE control. When using these macros, specifying a $NOGEN control to suppress the macro expansion printout will greatly improve the readability of your listing.

The RMX/80 Extensions diskette contains additional macros to aid in coding the extra portions of the configuration module required by the Disk File System. These macros are described in Chapter 7.

The macros provided on the Executive diskettes are STD, which builds Static Task Descriptors; XCHADR, which builds Initial Exchange Table entries; GENTD, which allocates RAM for Task Descriptors; and CRTAB, which builds the Create Table. Because GENTD and CRTAB both refer to items defined by the STD and XCHADR macros, GENTD and CRTAB must be coded after all the STD and XCHADR macros.

Before coding any of the macros, the user must issue two SET directives to initialize two counters, NEXCH and NTASK, to zero as follows:

```
NEXCH           SET  0
NTASK           SET  0
```

STD Macro. The STD macro builds a Static Task Descriptor. The format for STD is as follows:

    STD name, stklen, pri[,exch][,tdxtra]

Name is the symbolic name assigned to the procedure or subroutine associated with this Static Task Descriptor, and is the entry address of the task. The name must conform to the assembly language rules for symbols. (A symbol may be from one to six characters in length and must begin with a letter of the alphabet. The remainder of the name may be any combination of letters or digits, except that the second character should not be "Q" or "?".)

Stklen is the length (in bytes) of the stack for this task.

Pri is the priority of the task.

Exch is an optional field used to identify a default exchange to be associated with the task. If used, this field must specify the symbolic name of the exchange.

Tdxtra is a field reserved for future use and should not be coded.

Because the Initial Task Table is made up of a list of contiguous Static Task Descriptors, one STD macro must be given for each task to be initialized by RMX/80. This list of STD macros cannot contain any embedded instructions or data.

XCHADR Macro. The XCHADR macro builds an Initial Exchange Table entry. The format for XCHADR is as follows:

    XCHADR name

Name is the symbolic name assigned to the Exchange Descriptor associated with this Initial Exchange Table entry. The name must conform to the assembly language rules for symbols and should not begin with "RQ."

Because the Initial Exchange Table is a list of addresses of Exchange Descriptors, one XCHADR macro must be given for each exchange to be initialized by RMX/80.

This list cannot contain any embedded instructions or data.

GENTD Macro. The GENTD macro allocates a single block of RAM for the Task Descriptors to be generated when the system is initialized. The format for a call to GENTD is as follows:

    GENTD

GENTD refers to data defined by the STD macros; therefore, the call GENTD must follow the list of STD macros.

Operands are not permitted with the GENTD macro.

CRTAB Macro. The CRTAB macro builds the Create Table. The format for a call to CRTAB is as follows:

    CRTAB

CRTAB refers to data defined by the STD and XCHADR macros; therefore, the call to CRTAB must follow these macros.

Operands are not permitted with the CRTAB macro.

PL/M Configuration Module for Example Application.

```
CONFIGURATION$MODULE: DO;

/* FIRST DECLARE SOME SYMBOLIC VALUES TO MAKE LATER
   CODING EASIER. NOTE THAT MOST OF THESE ARE
   DISTRIBUTED ON THE SYSTEM DISK AND CAN BE
   $INCLUDED IN THE USER'S PROGRAM (SEE APPENDIX A).
 */

DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          MESSAGE$HEAD          ADDRESS,
          MESSAGE$TAIL          ADDRESS,
          TASK$HEAD             ADDRESS,
          TASK$TAIL             ADDRESS,
          EXCHANGE$LINK         ADDRESS) ';

DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          MESSAGE$HEAD          ADDRESS,
          MESSAGE$TAIL          ADDRESS,
          TASK$HEAD             ADDRESS,
          TASK$TAIL             ADDRESS,
          EXCHANGE$LINK         ADDRESS,
          LINK                  ADDRESS,
          LENGTH                ADDRESS,
          TYPE                  BYTE) ';

DECLARE TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
          DELAY$LINK$FORWARD    ADDRESS,
          DELAY$LINK$BACK       ADDRESS,
          THREAD                ADDRESS,
          DELAY                 ADDRESS,
          EXCHANGE$ADDRESS      ADDRESS,
          SP                    ADDRESS,
          MARKER                ADDRESS,
          PRIORITY              BYTE,
          STATUS                BYTE,
          NAME$PTR              ADDRESS,
          TASK$LINK             ADDRESS) ';
```

```
DECLARE STATIC$TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
        NAME(6)                 BYTE,
        PC                      ADDRESS,
        SP                      ADDRESS,
        STKLEN                  ADDRESS,
        PRIORITY                BYTE,
        EXCHANGE$ADDRESS        ADDRESS,
        TASK$PTR                ADDRESS) ';

DECLARE CREATE$TABLE LITERALLY 'STRUCTURE (
        TASK$POINTER            ADDRESS,
        TASK$COUNT              BYTE,
        EXCHANGE$POINTER        ADDRESS,
        EXCHANGE$COUNT          BYTE) ';

/**** STACK LENGTHS ****/
DECLARE   TH$STL     LITERALLY    '36';       /* RQTHDI */
DECLARE   FS$STL     LITERALLY    '40';       /* RQFMGR */
DECLARE   DB$STL     LITERALLY    '64';       /* RQADBG */
DECLARE   U1$STL     LITERALLY    '24';       /* USER1 */
DECLARE   U2$STL     LITERALLY    '28';       /* USER2 */
DECLARE   U3$STL     LITERALLY    '48';       /* USER3 */

/**** TASK PRIORITIES ****/
DECLARE   TH$PRI     LITERALLY    '112';      /* RQTHDI */
DECLARE   FS$PRI     LITERALLY    '130';      /* RQFMGR */
DECLARE   DB$PRI     LITERALLY    '140';      /* RQADBG */
DECLARE   U1$PRI     LITERALLY    '35';       /* USER1 */
DECLARE   U2$PRI     LITERALLY    '135';      /* USER2 */
DECLARE   U3$PRI     LITERALLY    '150';      /* USER3 */

/**** TASK STACKS ****/
DECLARE   TH$STK (TH$STL)        BYTE;        /* RQTHDI */
DECLARE   FS$STK (FS$STL)        BYTE;        /* RQFMGR */
DECLARE   DB$STK (DB$STL)        BYTE;        /* RQADBG */
DECLARE   U1$STK (U1$STL)        BYTE;        /* USER1 */
DECLARE   U2$STK (U2$STL)        BYTE;        /* USER2 */
DECLARE   U3$STK (U3$STL)        BYTE;        /* USER3 */

/**** TASK  DESCRIPTORS ****/
DECLARE   TH$TD      TASK$DESCRIPTOR;         /* RQTHDI */
DECLARE   FS$TD      TASK$DESCRIPTOR;         /* RQFMGR */
DECLARE   DB$TD      TASK$DESCRIPTOR;         /* RQADBG */
DECLARE   U1$TD      TASK$DESCRIPTOR;         /* USER1 */
DECLARE   U2$TD      TASK$DESCRIPTOR;         /* USER2 */
DECLARE   U3$TD      TASK$DESCRIPTOR;         /* USER3 */

/**** TASK EXTERNAL DECLARATIONS ****/
DECLARE   RQTHDI:            PROCEDURE EXTERNAL;             /* RQTHDI */
          END RQTHDI;
DECLARE   RQFMGR:            PROCEDURE EXTERNAL;             /* RQFMGR */
          END RQFMGR:
DECLARE   RQADBG:            PROCEDURE EXTERNAL;             /* RQADBG*/
          END RQADBG;
DECLARE   USER1:             PROCEDURE EXTERNAL;             /* USER1 */
          END USER1;
DECLARE   USER2:             PROCEDURE EXTERNAL;             /* USER2 */
          END USER2;
DECLARE   USER 3:            PROCEDURE EXTERNAL;             /*  USER3  */
          END    USER3;
```

```
/**** INITIAL TASK TABLE ****/
DECLARE   NTASK              LITERALLY '6';
DECLARE   ITT(NTASK)         STATIC$TASK$DESCRIPTOR DATA (


          /* STD FOR INPUT-OUTPUT TERMINAL HANDLER */
          'RQTHDI',          /* TASK NAME */
          .RQTHDI,           /* INITIAL P.C. (ENTRY POINT) */
          .TH$STK,           /* STACK POINTER */
          TH$STL,            /* STACK LENGTH */
          TH$PRI,            /* PRIORITY */
          .RQOUTX,           /* DEFAULT EXCHANGE */
          .TH$TD,            /* TASK DESCRIPTOR */


          /* STD FOR FREE SPACE MANAGER */
          'RQFMGR',          /* TASK NAME */
          .RQFMGR,           /* INITIAL P.C. */
          .FS$STK,           /* STACK POINTER */
          FS$STL,            /* STACK LENGTH */
          FS$PRI,            /* PRIORITY */
          .RQFSRX,           /* DEFAULT EXCHANGE */
          .FS$TD,            /* TASK DESCRIPTOR */


          /* STD FOR ACTIVE DEBUGGER */
          'RQADBG',          /* TASK NAME */
          .RQADBG,           /* INITIAL P.C. */
          .DB$STK,           /* STACK POINTER */
          DB$STL             /* STACK LENGTH */
          DB$PRI,            /* PRIORITY */
          .RQWAKE,           /* DEFAULT EXCHANGE */
          .DB$TD,            /* TASK DESCRIPROT */


          /* STD FOR USER TASK 1 */
          'USER1',           /* TASK NAME */
          .USER1,            /* INITIAL P.C. */
          .U1$STK,           /* STACK POINTER */
          U1$STL,            /* STACK LENGTH */
          U1$PRI,            /* PRIORITY /*
          .RQL2EX,           /* DEFAULT EXCHANGE */
          .U1$TD,            /* TASK DESCRIPTOR */


          /* STD FOR USER TASK 2 */
          'USER2',           /* TASK NAME */
          .USER2,            /* INITIAL P.C. */
          .U2$STK,           /* STACK POINTER */
          U2$STL,            /* STACK LENGTH */
          U2$PRI,            /* PRIORITY /*
          .USXCH2,           /* DEFAULT EXCHANGE */
          .U2$TD,            /* TASK DESCRIPTOR */


          /* STD FOR USER TASK 3 */
          'USER3',           /* TASK NAME */
          .USER3,            /* INITIAL P.C. */
          .U3$STK,           /* STACK POINTER */
          U3$STL,            /* STACK LENGTH */
          U3$PRI,            /* PRIORITY /*
          .USXCH3,           /* DEFAULT EXCHANGE */
          .U3$TD);           /* TASK DESCRIPTOR */
```

```
/**** EXCHANGE DESCRIPTORS ****/
DECLARE   RQINPX      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* RQINPX */
DECLARE   RQOUTX      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* RQOUTX */
DECLARE   RQWAKE      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* RQWAKE */
DECLARE   RQDBUG      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* RQDBUG */
DECLARE   RQALRM      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* RQALRM */
DECLARE   RQFSRX      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* RQFSRX */
DECLARE   RQFSAX      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* RQFSAX */
DECLARE   USXCH1      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* USXCH1 */
DECLARE   USXCH2      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* USXCH2 */
DECLARE   USXCH3      EXCHANGE$DESCRIPTOR         EXTERNAL;  /* USXCH3 */
DECLARE   RESP$EX     EXCHANGE$DESCRIPTOR         EXTERNAL;  /*RESP$EX*/
/**** INTERRUPT EXCHANGE DESCRIPTORS ****/
DECLARE   RQL6EX      INT$EXCHANGE$DESCRIPTOR     PUBLIC;    /* LEVEL 6 */
DECLARE   RQL7EX      INT$EXCHANGE$DESCRIPTOR     PUBLIC;    /* LEVEL 7 */
DECLARE   RQL2EX      INT$EXCHANGE$DESCRIPTOR     PUBLIC;    /* LEVEL 2 */


/**** INITIAL EXCHANGE TABLE ****/
DECLARE   NEXCH       LITERALLY '14';
DECLARE   IET(NEXCH)                 ADDRESS DATA (
          .RQINPX,                   /* RQINPX */
          .RQOUTX,                   /* RQOUTX */
          .RQWAKE,                   /* RQWAKE */
          .RQDBUG,                   /* RQDBUG */
          .RQALRM,                   /* RQALRM */
          .RQFSRX,                   /* RQFSRX */
          .RQFSAX,                   /* RQFSAX */
          .USXCH1,                   /* USXCH1 */
          .USXCH2,                   /* USXCH2 */
          .USXCH3,                   /* USXCH3 */
          .RQL6EX,                   /* LEVEL 6 INTERRUPT */
          .RQL7EX,                   /* LEVEL 7 INTERRUPT */
          .RQL2EX,                   /*LEVEL 2 INTERRUPT*/
          .RESP$EX);                 /*RESP$EX*/
/**** CREATE TABLE ****/
DECLARE   RQCRTB                     CREATE$TABLE PUBLIC DATA (
          .ITT,
          NTASK,
          .IET,
          NEXCH);

/**** PUBLIC DATA ****/
DECLARE   RQRATE                     ADDRESS PUBLIC DATA (56);

END CONFIGURATION $ MODULE;
```

Assembly-Language Configuration Module for Example Application.

```
          NAME      CONFIG
          CSEG
;   INCLUDE MACRO DEFINITIONS FROM PRODUCT DISKETTES
$INCLUDE   (:F2:STD.MAC)
$INCLUDE   (:F2:XCHADR.MAC)
$INCLUDE   (:F2:CRTAB.MAC)
$INCLUDE   (:F2:GENTD.MAC)
NTASK   SET       0
NEXCH   SET       0
;
;   BUILD THE INITIAL TASK TABLE (ITT)
;
          STD       RQTHDI,36,112,RQOUTX
          STD       RQFMGR,40,130,RQFSRX
          STD       RQADBG,64,140,RQWAKE
          STD       USER1,24,35,RQL2EX
          STD       USER2,28,135,USXCH2
          STD       USER3,48,150,USXCH3
;
;   ALLOCATE THE TASK DESCRIPTORS
;
          GENTD
;
;   BUILD THE INITIAL EXCHANGE TABLE (IET)
;
          XCHADR    RQINPX
          XCHADR    RQOUTX
          XCHADR    RQWAKE
          XCHADR    RQDBUG
          XCHADR    RQALRM
          XCHADR    RQL6EX
          XCHADR    RQL7EX
          XCHADR    RQFSRX
          XCHADR    RQFSAX
          XCHADR    RQL2EX
          XCHADR    USXCH1
          XCHADR    USXCH2
          XCHADR    USXCH3
;         XCHADR    RESP$EX
;   BUILD CREATE TABLE
;
          CRTAB
;
;   DEFINE PUBLIC DATA
;
          PUBLIC    RQRATE
RQRATE:   DW        56
          END
```

Linking and Locating

After you have prepared your configuration module (and controller-addressable memory module for the Disk File System), you are now ready to link together the relocatable object modules for the RMX/80 Nucleus and extension tasks, your user tasks, and your configuration module (and controller-addressable memory, or CAM module for DFS) and prepare them for running by using the ISIS-II LINK and LOCATE programs. The following paragraphs supply all the information you need about RMX/80 in order to link and locate your system. For a complete description of the LINK and LOCATE programs, refer to ISIS-II System User's Guide, order number 9800306.

Linking. In order to use LINK, you must have all your user taks and your con-
figuration and, in configuration using the DFS, CAM modules in ISIS-II diskette
libraries. These libraries are to be named in the LINK command in the order listed
below. (This order is important; deviations from it may render RMX/80 unable to
resolve external references.)

1. RMX/80 START module
2. Configuration module, RMX/80 extension libraries, and user task libraries
3. RMX820.LIB, RMX830.LIB, or RMX810.LIB
4. UNRSLV.LIB
5. PLM80.LIB

The RMX/80 product diskette supplies all these libraries except for the configura-
tion module and user task libraries. (Refer to Appendix A for descriptions of these
libraries.) UNRSLV.LIB is supplied to resolve external references in the system.
RMX/80 requires portions of the PL/M-80 library, PLM80.LIB, which is also sup-
plied on the RMX/80 diskette. The chapters on the RMX/80 extensions specify
which libraries must be linked in to use the extensions.

If the object code of the PL/M or assembly-language configuration module for our
example application system has been placed in a library called USRCOD.LIB along
with the rest of the user-supplied object code, and if the RMX/80 diskette and all
user files are on drive n, the following commands can be used to link the system (for
the 80/20 version):

```
LINK :Fn:RMX820.LIB (START), &
      :Fn:USRCOD.LIB, &
      :Fn:ADB820.LIB, &
      :Fn:PDB820.LIB, &
      :Fn:THI820.LIB, &
      :Fn:THO820.LIB, &
      :Fn:TSK820.LIB, &
      :Fn:RMX820.LIB, &
      :Fn:UNRSLV.LIB, &
      :Fn:PLM80.LIB TO :Fn: SYSTEM.LNK
```

The RQSUSP, RQRESM, RQDTSK, and RQDXCH operations are optional and
are not part of the basic RMX/80 Nucleus. These operations are automatically link-
ed into a system only if they are called by a user task or tasks. If these operations are
not called in any user tasks and you want to use them for debugging, you must link
them into the application code by adding the following files (for the iSBC 80/20 ver-
sion) to the list of files given to the LINK command.

```
:Fn:RMX820.LIB(SUSPND)
:Fn:RMX820.LIB(RESUME)
:Fn:RMX820.LIB(DLTASK)
:Fn:RMX820.LIB(DLEXCH)
```

For the iSBC 80/30 and 80/10 versions, use the LINK list given for the 80/20 ver-
sion, with the substituting of the string "830.LIB" or "810.LIB," respectively, for
the string "820.LIB" wherever it appears.

You can generally save time by linking modules into the system one by one rather
than all at once. During testing and debugging, you may wish to link modules into
the system as they are debugged. You can make the linking process even more effi-
cient by linking and locating the already-debugged portion of your system separately

from the portion still under test. The output from the LOCATE program will supply a list of unresolved external references. You can resolve these by linking each of the two output modules from LOCATE to the PUBLICS of the other, as described in the ISIS-II System User's Guide, 9800306. This way, each time modifications are made, you need re-LINK and re-LOCATE only the code being debugged.

If your code includes a subroutine shared by several tasks, you must either make it reentrant or supply a separate copy for each task. An easy way to do the latter is to link the subroutine separately to each of the tasks that calls it.

Locating. The following command will locate our example application system.

```
LOCATE :Fn:SYSTEM.LNK  TO  SYSTEM.LOC
CODE(0)  DATA (3800H)  STACKSIZE(0)  START(0)
```

The starting address for code in most RMX/80 systems is specified as zero. Also, the stack size is specified as zero, since each task has its own stack.

The starting address given above for the data segment (3800H) is the default starting address for on-board RAM on the iSBC 80/20. This default starting address is 3000H for the iSBC 80/20-4; 6000H for the iSBC 80/30; and 3C00H for the iSBC 80/10.

On the iSBC 80/20, 80/20-4, and 80/30, the starting address can be re-wired to one of several other values. You will find it necessary to modify the RAM starting address if the size of your code segment (which is typically in ROM) exceeds the default starting address. Alternatively, you may elect to use some other RAM starting address if your application's RAM requirements exceed the amount of RAM storage on your iSBC 80.

NOTE

RMX/80 uses the following Absolute memory locations for Restart Vectors:

80/30	80/20	80/10
24H - 2EH	28H - 2AH	28H - 2AH
34H - 36H		38H - 3AH
3CH - 3EH		

Locations 28H - 2AH are used by the Active Debugger for Execution Breakpoints. Other locations are for Interrupt Restart Vectors.

Your RMX-based system will normally be LOCATED beginning at memory location 0000, and the LOCATER will issue warnings of memory conflicts at the Restart Locations. RMX has allowed space for this in its code and the messages can be ignored.

If you choose *not* to LOCATE at zero, you should LOCATE at 40H or higher and no other code should use the memory locations mentioned above. You must also provide code at location 0000 to transfer control to the RMX/80 START module when the system is RESET.

Unresolved External References

The RMX/80 Nucleus declares all interrupt exchanges except RQL1EX (used for the system clock) as EXTERNAL. This is because user interrupt tasks must define the exchanges as needed. Any of these interrupt exchanges that are not used in a system,

and are therefore not declared in user code, are treated as unresolved external references by the Linker, the Locater, ICE-80 and ICE-85. Messages issued by these products that refer to unused interrupt exchanges can be considered as warnings and ignored. The messages issued by the various products are:

Linker: UNRESOLVED EXTERNAL NAMES
                      xxxxxx
                      xxxxxx
                      etc.

Locater: UNRESOLVED EXTERNAL REFERENCE AT xxxxH
                      (two messages for each exchange)

ICE-80:   ERR=069

ICE-85:   *WARNING UNSATISFIED EXTERNALS

You should check the messages to be certain that none of them refers to anything other than an unused interrupt exchange. Appendix J shows one way to "tie off" references to unused interrupt exchanges in a configuration module.

Testing and Debugging

The Intellec microcomputer development system, with its ICE-80 and ICE-85 options, provides a convenient means for testing and debugging your RMX/80 application system. The features of this system are covered in the set of manuals listed in the Preface to this manual.

However, the unique operational structure of a real-time system makes certain additional debugging features worthwhile. The RMX/80 Debugger, especially tailored for systems based on RMX/80, provides such features. Chapter 6 supplies complete instructions and suggestions for using the Debugger to test and debug your application system.

General Description

The Terminal Handler provides real-time asynchronous I/O between an operator's terminal and tasks running under the RMX/80 Nucleus. The Terminal Handler provides a line-edit capability similar to that of ISIS-II (see ISIS-II User's Guide, order number 9800306) and an additional type-ahead feature. The type-ahead feature allows the terminal operator to enter (or continue entering) up to 63 characters (with no echo) even when there is no current request for input.

The Terminal Handler completely relieves user tasks from the duties required to perform physical I/O with a terminal. Instead, user tasks perform input at the level of a logical line, taking input data from the Terminal Handler's line buffer. Output may be at the level of a single character, a logical line, or a set of lines.

The line-edit feature assures the user task of receiving relatively "clean" input from the terminal. For example, by using the RUBOUT control, the operator can correct a typing error before it is passed to the user task. In addition, the Terminal Handler output display (printout on a Teletypewriter) provides an echo of all input characters entered in the line buffer.

Variations

There are three separate versions of the Terminal Handler, one for each of the iSBC 80/20, 80/30, and 80/10. Additionally, each version can be configured for either input-output or output only, and for either the full Terminal Handler or the minimal Terminal Handler. The principal differences between these configurations are displayed in a table at the end of this chapter.

Use Environment

The Terminal Handler (in any of its configurations) is designed to accommodate a variety of different terminal models. Terminals may be of the CRT display or teletypewriter design with baud rates ranging from 110 to 9600, asynchronous. Notice, however, that the Terminal Handler directly supports only one terminal in a system. The Terminal Handler supports a maximum input line length of 132 characters.

Memory and Hardware Requirements

Memory requirements range from approximately 3000 bytes of code space and 900 bytes of data space for the full input-output Terminal Handler to less then 200 bytes of code space and 100 bytes or less of data space for the minimal output-only Terminal Handler. Refer to Appendix D for exact byte counts.

The Terminal Handler uses interrupt level 6 for input and 7 for output. In the iSBC 80/20 and 80/30 versions, these interrupts are routed through the 8259 Programmable Interrupt Controller. In the iSBC 80/10 version, the RMX/80 software pseudo-interrupt levels are used. The interrupts originate in the 8251 Programmable Communications Interface (USART), which is used for all I/O on the iSBC 80.

In the iSBC 80/10 version, the Terminal Handler supplies the interrupt polling routines for input and output and the RQSETP calls to set them up, so that the user task need not be concerned with these routines. Pseudo-interrupt level 6 is used for input, and level 7 for output.

On the iSBC 80/20 or 80/30, the full Terminal Handler can provide an optional automatic baud rate search. (This option is described under "Using the Terminal Handler" later in the chapter.) On the iSBC 80/20 and 80/30, the Terminal Handler uses Counter 2 of the 8253 Interval Timer to generate baud rate timing for the 8251 USART.

How the Terminal Handler Operates

Input and Output

A logical line is a series of characters terminated by a break sequence. In the full Terminal Handler, a break sequence may be an Escape character (ESC), a Line Feed character (LF), or the sequence Carriage Return-Line Feed (CR,LF). In the minimal Terminal Handler, the only break character is a Carriage Return (CR). The Terminal Handler automatically generates a Line Feed after a Carriage Return. However, if a Carriage Return happens to fill the last (132nd) buffer position, there will be no room to generate the Line Feed. The Terminal Handler recognizes a break sequence in this situation; application tasks that process the input lines should be programmed to take this possibility into account.

Input operations are normally performed one logical line at a time. When a user task is ready to accept input, it sends a read request message to the Terminal Handler. This message tells the Terminal Handler the buffer address where it is to place the input data and the number of characters the user task is prepared to receive. The Terminal Handler responds by moving the specified number of characters from the current logical line into the user buffer.

The minimal Terminal Handler will accept enough input characters to fill the current outstanding read request (or until a carriage return). If the current read request is filled before a carriage return is input, the minimal Terminal Handler issues a BEL character and accepts no more input until the next read request.

The full Terminal Handler will accept enough input characters to fill the current outstanding read request (or until a break sequence), plus the 63 characters it will accept into its type-ahead buffer. When the type-ahead buffer is full, the Terminal Handler will output a BEL character and accept no more input until the type-ahead buffer is cleared by a read request. A requesting task does not need to accept the entire logical line at one time. For example, the task might read only five characters of the logical line at a time. (This technique can be used to conserve RAM.) The task can detect the end of a line by checking for the break sequence.

In both minimal and full input-output versions, the Terminal Handler indicates the completion of the read operation by returning the read request message with a field that indicates the actual number of characters placed in the user buffer. This is helpful when the logical line does not contain enough characters to satisfy the count requested for the read.

To write a logical line to the terminal, the user task first builds the desired line in a RAM buffer. The task then sends a write request message to the Terminal Handler. This message tells the Terminal Handler the length and address of the output line. To the user task, the output operation appears complete at this point. However, the

Terminal Handler must still perform the physical output. Completion of physical output is indicated by returning the write request message with a status code to let the task know whether the operation is successful. The user task must not attempt to build another output line in the same RAM locations until the output operation is complete. You may, of course, initiate output operations from other RAM locations before the preceding operations are complete.

Line Editing

As an operator enters keystrokes (provided a read request message is available at the Terminal Handler input exchange) the corresponding characters are placed in the Terminal Handler's line (input) buffer. The Terminal Handler then echoes a copy of each character back to the terminal. (However, echo output must be enabled as described under "Read Request Message" later in this section.) Remember that the terminal is both an input and an output device. Each keystroke is an input to the system. Any character displayed on the terminal is an output from the system. In most cases, the echo is returned to the terminal so quickly that the operator is not aware that the keystroke does not directly affect the display line.

If no read request message is currently available, none of the preceding steps takes place. In the full Terminal Handler, up to 63 characters are stored in the type-ahead buffer. When a read request message becomes available, the characters are transferred from the type-ahead buffer to the line buffer, and the echo is then performed. However, the minimal Terminal Handler has no type-ahead buffer; so when the minimal Terminal Handler is used, characters that are input when there is no read request message are simply lost.

Because of error corrections, there is not always a direct correspondence between the contents of the Terminal Handler's input buffer and the terminal's display line. For example, RUBOUT causes the Terminal Handler to delete the previous character from its input buffer. At the same time, the Terminal Handler echoes a second copy of the character to the display to assure the operator that the correction has actually occurred. The following example illustrates the difference between the display and the input buffer. Assume that the operator enters TARFET and corrects it to TARGET using three RUBOUTs:

```
DISPLAY LINE:   TARFETTEFGET
INPUT BUFFER:  TARGET
```

(The control-R command, described later in this chapter, may be used by the operator to generate a clean copy of the input buffer.) Maintaining both the input buffer and the terminal display are two more services provided by the Terminal Handler.

Terminal Handler Exchanges

NOTE

In what follows, there are numerous references to RQALRM, RQDBUG, and the Debugger. The reader should bear in mind that these features are not operative on the minimal Terminal Handler.

Figure 4-1 graphically illustrates the relationships between a user task and the Terminal Handler. The illustration is simplified for clarity.
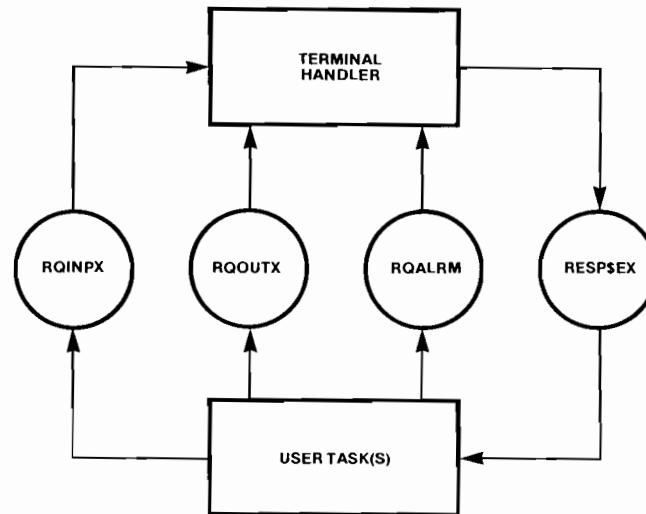
Figure 4-1. Terminal Handler Exchanges for Full Input-Output Versions

Circles represent exchanges. Although the RQINPX and RQOUTX exchanges are the primary user task interfaces with the Terminal Handler, an additional high-priority output exchange is also provided in the full Terminal Handler. The RQALRM exchange functions the same as the RQOUTX exchange, with two exceptions: its priority is higher, and output to it cannot be blocked by the control-S command (see "Output Controls" later in this chapter). Also, the RQALRM exchange can optionally generate an alarm message line before it displays the user output. The alarm message is displayed as follows:

     *** ALARM

The alarm message also sends two BEL characters to the terminal. The BEL characters sound the terminal's audible alarm, if it has one. Notice that output messages may be queued at both the RQALRM and RQOUTX exchanges. Since the RQALRM exchange has the higher priority, alarm messages are always displayed before all other output, including echo output. (This can cause echo characters to be interspersed with a message coming out from RQALRM.) Also, remember that all messages sent to an exchange are processed on a first-in/first-out basis.

NOTE

Since RQALRM has the higher priority, it will preempt a display in progress through RQOUTX.

Debugger Exchanges

Two additional exchanges, illustrated in figure 4-2, provide an interface between the Terminal Handler and the RMX/80 Debugger (described in Chapter 6). The first of these is the RQWAKE exchange, which is used to invoke the Debugger. A message of type CNTRLCTYPE is sent to this exchange only when the operator enters a control-C command.
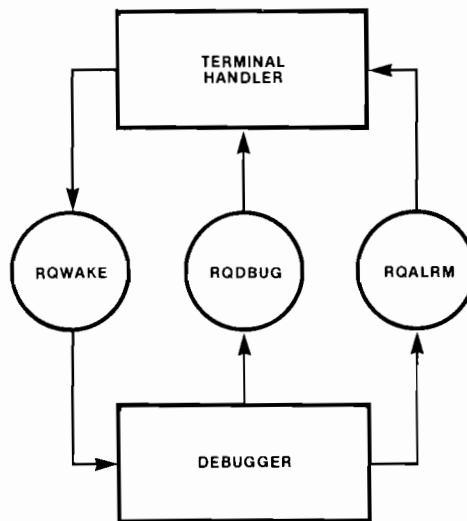
Figure 4-2. Terminal Handler-Debugger Exchanges

In a system that includes the Debugger, the control-C causes the following operations to occur:

•   The Debugger is activated.

•   The RQINPX exchange is locked out and remains locked out until the Debugger is exited via the QUIT (Q) command. While the Debugger is active, all input is directed to the Debugger and is subject to the Debugger's input conventions.

•   The Debugger instructs the Terminal Handler to clear any residual data remaining in its buffers.

•   The Debugger outputs a prompt character via the Terminal Handler.

In a system configured without the Debugger, entering a control-C causes a message to be sent to the RQWAKE exchange. Such a system has no built-in response to this message, thus allowing you to use it for your own purposes.

Because the RQINPX exchange is locked out while the Debugger is active, the RQDBUG exchange is provided for Debugger input.

NOTE

The direction of the arrows indicates the direction of messages sent to and received from exchanges. Response exchanges are not shown.

Notice that the Debugger uses the Terminal Handler's high-priority RQALRM exchange for output. Although the Debugger blocks normal input by locking the RQINPX exchange, the RQOUTX and RQALRM exchanges continue to operate normally. If desired, the operator can block output via RQOUTX with a control-S command. RQALRM exchange output cannot be blocked, since the Debugger requires use of this exchange.

Using the Terminal Handler

The Terminal Handler differs from other RMX/80 extensions in that it involves both a programming interface and an operator interface. These two concerns overlap when the optional baud rate search (available on the full input-output Terminal Handler for the iSBC 80/20 or 80/30) is used. The primary programming interface between user tasks and the Terminal Handler is via read and write request messages. For the operator interface, the input-output versions of the Terminal Handler supply a set of one-character control commands.

Terminal Baud Rate

Because different terminals transmit data at different speeds, the Terminal Handler references the PUBLIC variable RQRATE. The value of RQRATE informs the Terminal Handler at what baud rate the terminal operates. RQRATE must correspond to the terminal baud rate as specified in table 4-1.

Table 4-1. Terminal Handler RQRATE Values

ACTUAL BAUD RATE	RQRATE VALUE (iSBC 80/20 and 80/10)	RQRATE VALUE (iSBC 80/30)
9600	7	8
4800	14	16
2400	28	32
1200	56	64
600	112	128
300	224	256
150	448	512
110	611	700

Supplying Fixed Value at Configuration. The user must provide a non-zero value for RQRATE. (This is optional for the iSBC 80/20 and 80/30 versions of the full input-output Terminal Handler; see below.) This is done by declaring RQRATE as a PUBLIC variable in a user module (preferably the configuration module) and then initializing it (according to table 4-1) as in the following assembly language code.

```
                PUBLIC  RQRATE
RQRATE:  DW       7
```

For the iSBC 80/10 version, in addition to the proper setting of RQRATE, certain hardware considerations are also involved in baud rate selection. Refer to Appendix F for details.

Automatic Baud Rate Search. With the iSBC 80/20 and 80/30 versions of the full input-output Terminal Handler, you may optionally specify an automatic baud rate search by assigning RQRATE the value zero. Alternatively, you may omit all references to RQRATE, and the 80/20 and 80/30 UNRSLV.LIB will provide a default value of 0 for RQRATE. Using the baud rate search allows you to use terminals with different baud rates, since the rate is determined each time the system is initialized. Without the baud rate search, you must reconfigure the system to change baud rates.

In order to activate a rate search, the operator must enter a string of U's, less than one second apart, each time the system is restarted. The alternating bit pattern of the uppercase U (0101 0101) allows the Terminal Handler to determine the baud rate of the sending terminal. The operator must repeat this entry until the U's are echoed normally, indicating that the terminal and the system are synchronized.

The need to initialize the baud rate is not obvious to an inexperienced operator. If you elect to use the rate search option, you must be certain that anyone who uses the terminal understands this procedure and can perform it correctly.

Read and Write Request Messages

Task communicate with the Terminal Handler by means of two types of messages: read request and write request. A read request message consists of a standard RMX/80 message heading and the REMAINDER portion that is similar to an ISIS-II READ system call. Likewise, the write request message has a REMAINDER portion similar to an ISIS-II WRITE system call. A read or write operation is performed by sending a request message to the appropriate exchange, as described in the following sections. Standard PL/M declarations of Terminal Handler messages can be obtained by using the INCLUDE file THMSG.ELT.*

Read Request Message. The read request message is generally sent to the RQINPX exchange, although in some cases it may be sent to RQDBUG (see description of TYPE field for this message). The read request message performs two important functions:

• It initiates the transfer of input characters from the type-ahead buffer into the Terminal Handler's line buffer. (Remember that the minimal Terminal Handler has no type-ahead buffer.) This function is especially important because input data is echoed back to the terminal's display only from the line buffer. Characters are not placed in the line buffer unless there is an outstanding read request. This deferred echoing can be quite useful, since it allows the operator to continue entering data while data is being written to the display. However, it can be confusing for an operator to make a lengthy "blind" entry (one that is not echoed within a reasonable time). To prevent such delays, it is advisable that a task which requests input wait immediately thereafter at the appropriate response exchange.

• It signals the Terminal Handler that the user task is ready to accept data. However, before releasing any data into the user task's buffer, the Terminal Handler waits until it has a complete logical line or until all 132 characters of its line buffer are filled. The Terminal Handler forces a break sequence when the line buffer is filled so that subsequent data is not lost.

When the Terminal Handler has processed the request, it returns the read request message with a status code that indicates either successful completion of the transfer, or an unacceptable message type.

The format of the read request message is shown in figure 4-3. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by the Terminal Handler when it returns the message.

Those fields that have specific meanings in the read request message are described in the following paragraphs. (For general information, refer to "Message Coding" in Chapter 2.)

LENGTH is 17.

In the full Terminal Handler, TYPE must be one of the following types: READ$TYPE (8), CLR$RD$TYPE (9), or LAST$RD$TYPE (10). In the minimal Terminal Handler, READ$TYPE must be used.

- READ$TYPE is the most commonly used type of read request. In the full Terminal Handler, it allows a read using type-ahead. When directed to the RQDBUG exchange, this type has the additional function of locking out the RQINPX exchange, enabling you to send input to the Debugger via the RQDBUG exchange without the risk that the input will be interpreted as data by the executing task.

- CLRRDTYPE clears any type-ahead and then reads. This is useful when an emergency interrupt question is directed to the terminal (and an immediate response is expected). Since this event is unpredictable, there could be unwanted data in the type-ahead buffer, and this data would be treated as a response to the question. CLRRDTYPE insures that the type-ahead buffer is emptied so that you are sure you get the desired response rather than any residual type-ahead data. Use of the RQDBUG exchange with this message type should be considered reserved for the Debugger unless you are writing your own debugging or other high-priority routine.

- LASTRDTYPE allows a read using type-ahead. When directed to the RQDBUG exchange, this type has the additional function of unlocking the RQINPX exchange. This type should be considered reserved for the Debugger unless you are writing a debugging routine of your own. When directed to the RQINPX exchange, LASTRDTYPE functions the same as READ$TYPE.

HOME EXCHANGE is not used by the Terminal Handler.

RESPONSE EXCHANGE must specify the address of the user-defined exchange where the requesting task waits for a response from the Terminal Handler.

When the Terminal Handler returns this message, it sets STATUS to 0 to indicate normal completion of the read. STATUS is set to BAD$COMMAND (18) if the user specified a message type other than READ$TYPE, CLR$RD$TYPE, or LAST$RD$TYPE (if other than READ$TYPE, in the case of the minimal Terminal Handler). In this case, no data is transferred from the line buffer to the user buffer. The read request must be re-issued to correct this non-fatal error. ACTUAL is undefined when an error message is returned.

BUFFER ADDRESS must specify the RAM address where the Terminal Handler is to store the data resulting from the read request. Break sequence characters are considered part of the data and are placed in the buffer.

COUNT must specify the number of characters to be read. The maximum value is 132.

0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
11	BUFFER ADDRESS*
13	COUNT*
15	ACTUAL
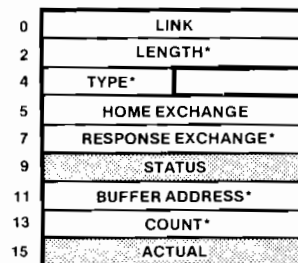
Figure 4-3. Read Request Message

In ACTUAL, the Terminal Handler supplies the actual number of characters transferred from the line buffer to the user buffer when it returns the request message to indicate completion of the read. This value will be less that the COUNT field when the line buffer does not contain enough characters to fill the user buffer. Characters located in the buffer beyond BUFFER ADDRESS + ACTUAL should be considered undefined.

Write Request Message. The write request message is sent to either the RQOUTX exchange or the RQALRM exchange. (See description of TYPE field for this message.) This message initiates a transfer of data from the user task's buffer to the terminal. The Terminal Handler returns the write request message with a status code that indicates either successful completion of the transfer, or an unacceptable message type.

The format of the write request message is shown in figure 4-4. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by the Terminal Handler when it returns the message.

Those fields that have specific meanings in the write request message are described in the following paragraphs. (For general information, refer to "Message Coding" in Chapter 2.)

LENGTH is 17.

In the full Terminal Handler, TYPE must be either WRITE$TYPE (12) or ALARM$TYPE (11). In the minimal Terminal Handler, WRITE$TYPE must be used.

- WRITE$TYPE directs the Terminal Handler to write out the specified bytes. This type is valid for either the RQOUTX or RQALRM exchange.

- ALARM$TYPE causes the Terminal Handler to write an alarm message (CR, LF, *, BEL, *, BEL, * ALARM, CR, LF) followed by the specified bytes. This type is valid only for the RQALRM exchange. (Recall that in the minimal Terminal Handler, RQALRM does not exist.)

Note that both WRITE$TYPE and ALARM$TYPE are valid for the RQALRM exchange. This allows you to write a single alarm message followed by a number of lines of text rather than have an alarm message before each line. Remember that output to the RQALRM exchange has priority over output to the RQOUTX exchange. ALARM$TYPE is not valid for RQOUTX. The RQALARM output will not be written to the terminal, however, until any RQOUTX message in progresss has been completed.

0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
11	BUFFER ADDRESS*
13	COUNT*
15	ACTUAL
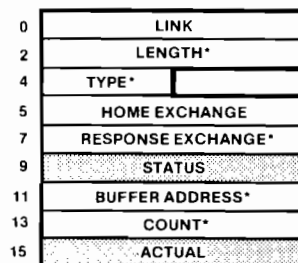
Figure 4-4. Write Request Message

RESPONSE EXCHANGE must specify the address of the user-defined exchange where the requesting task waits for a response from the Terminal Handler.

When the Terminal Handler returns this message, it sets STATUS to 0 to indicate normal completion of the write. STATUS is set to BAD$COMMAND (18) if the user specified a message type other than WRITE$TYPE or ALARM$TYPE (if other than WRITE$TYPE, in the case of the minimal Terminal Handler). No data is transferred from the user buffer to the terminal. The write request must be re-issued to correct this non-fatal error. ACTUAL is undefined when an error message is returned.

BUFFER ADDRESS must specify the RAM address from which the Terminal Handler is to write out data.

COUNT must specify the number of bytes to be written out. It is the user's responsibility to ensure that COUNT does not exceed the length of the buffer.

In ACTUAL, the Terminal Handler supplies the number of bytes written. If STATUS is zero, ACTUAL will always be equal to COUNT.

Example. The following PL/M code illustrates the use of read and write request messages. This example is intended to be a succinct illustration of the essential use of the Terminal Handler for input and output; it is not a complete, executable user task. However, this code has been derived (with some simplifications) from the example real-time clock system that is presented and described in full in Appendix J. Appendix J should be helpful as an illustration of the use of the Terminal Handler in a complete, executable RMX/80 application system.

```
/* MISCELLANEOUS "MACROS" */
DECLARE FOREVER LITERALLY 'WHILE 1';
DECLARE WRITE$TYPE LITERALLY '12';
DECLARE READ$TYPE LITERALLY '8';
DECLARE CR LITERALLY '0DH';
DECLARE LF LITERALLY '0AH';

/* TERMINAL HANDLER REQUEST MESSAGES */
DECLARE (COMMAND, REPLY) STRUCTURE (
    LINK                      ADDRESS,
    LENGTH                    ADDRESS,
    TYPE                      BYTE,
    HOME$EXCHANGE             ADDRESS,
    RESPONSE$EXCHANGE         ADDRESS,
    STATUS                    ADDRESS,
    BUFFER$ADR                ADDRESS,
    COUNT                     ADDRESS,
    ACTUAL                    ADDRESS);

/* TEXT OF MESSAGES (BUFFERS) */
DECLARE SIGNON$MSG(*) BYTE DATA(
        CR, LF, 'RTCLOK: VERSION 1.0',CR,LF,LF,
        'ALL TIMES ARE 24-HOUR FORMAT: HH:MM:SS',CR,LF,
        '    09:22:30 = 9:22:30 AM',CR,LF,
        '    17:30:00 =5:30 PM',CR,LF,
        '    00:00:00 = MIDNIGHT',CR,LF,
        '    24:00:00 = NO TIME - CLOCK IS OFF',CR,LF);
DECLARE COMMAND$MSG(81) BYTE;
```

```
/* TERMINAL HANDLER AND RESPONSE EXCHANGES */
DECLARE (RQOUTX,RQINPX) EXCHANGE$DESCRIPTOR EXTERNAL;
DECLARE (WRITEX,READEX) EXCHANGE$DESCRIPTOR;

CONSOL: PROCEDURE PUBLIC;
DECLARE SCRATCH BYTE;
    /* INITIALIZATION SEGMENT */
    CALL RQCXCH (.WRITEX); /* BUILD WRITE RESPONSE EXCH */

    /* BUILD AND SEND SIGNON MESSAGE TO TERMINAL */
    REPLY.LENGTH = SIZE (REPLY);
    REPLY.TYPE = WRITE$TYPE;
    REPLY.RESPONSE$EXCHANGE = .WRITEX;
    REPLY.BUFFER$ADDR = .SIGNON$MSG;
    REPLY.COUNT = SIZE(SIGNON$MSG);
    CALL RQSEND (.RQOUTX,.REPLY);

    /* BUILD COMMAND MESSAGE */
    CALL RQCXCH(.READEX); /* BUILD READ RESPONSE EXCH */
    COMMAND.LENGTH = SIZE (COMMAND);
    COMMAND.TYPE = READ$TYPE;
    COMMAND.RESPONSE$EXCHANGE = .READEX;
    COMMAND.BUFFER$ADR = .COMMAND$MSG;
    COMMAND.COUNT = SIZE(COMMAND$MSG);

    /* MAKE SURE SIGNON IS COMPLETE */
    SCRATCH = RQWAIT(.WRITEX,0);
    /* END OF INITIALIZATION SEGMENT*/

    /* LOOP TO ACCEPT COMMANDS FROM TERMINAL*/
    DO FOREVER;
        CALL RQSEND(.RQINPX,.COMMAND); /* REQUEST INPUT/*
        SCRATCH = RQWAIT(.READEX,0); /* WAIT FOR T.H. TO FINISH */

        /* USER CODE TO PROCESS THE DATA NOW
           IN COMMAND$MSG GOES HERE */

    END; /* OF DO FOREVER */
END CONSOL;
```

Control Commands

The full input-output Terminal Handler recognizes twelve one-character control commands; the minimal input-output Terminal Handler recognizes six of them. (No control commands are operative in the output-only versions.) The control commands fall into three general categories: line break control, line-edit control, and output control. The control commands have special meanings to the Terminal Handler, and are not echoed to the terminal. The Terminal Handler puts all other characters into its line buffer and echoes them back to the terminal.

The full Terminal Handler uses a table (RQCTAB) to recognize control characters. When configuring the system, you can alter this table to use different key codes that may be generated more easily by some terminals. For instance, if your terminal supplies BREAK and cursor keys that require only one keystroke, you may use these keys instead of control characters that require two keystrokes. Since the Terminal Handler is ASCII-based, it does not use codes in the range 80H to 0FFH, inclusive. Thus, if you wish to disable a particular control command, you need only assign the character a value greater than or equal to 80H. The control table is discussed further following the descriptions of the control commands.

In the minimal Terminal Handler, the values given in table 4-2 (for the commands recognized by the minimal Terminal Handler) are recognized; but these values cannot be altered, since no control table exists for the minimal Terminal Handler.

To enter any of the control commands except CR, LF, ESC, or RUBOUT, strike the designated letter key while holding down the control key.

NOTE

Because different terminals have different features, the terminology used in the following descriptions may not apply directly to your terminal. For example, some terminals may have a DELETE or BACKSPACE key rather than a RUBOUT key. Refer to the manufacturer's literature for details concerning your terminal.

Line Break Controls.

CR (Carriage Return). The Carriage Return is a line break character, generated when the operator strikes the Carriage Return key. The character is placed in the line buffer and echoed to the terminal (its representation in the display depends on your terminal). If there is room in the buffer, the Terminal Handler adds a Line Feed character to the buffer and echoes it to the terminal.

LF (Line Feed). The Line Feed is a line break character in the full Terminal Handler; in the minimal Terminal Handler it is stored as a regular character and does not signify a line break. The character is placed in the line buffer and echoed to the display.

ESC (ESCAPE). ESCAPE is a line break character. It is placed in the line buffer but is not echoed on the display. In the minimal Terminal Handler, ESC is not recognized as a line break control, but is treated as an ordinary character.

Control-Z. The control-Z is an end-of-input character that can be used to simulate an end-of-file from the keyboard. Its exact meaning depends on your program logic. When a read request encounters a control-Z anywhere in the input line, the line buffer is cleared, and no data is moved into the user buffer. Instead, the Terminal Handler returns the read request message with its ACTUAL field set to zero. Your program can interpret this as an actual end-of-input and stop issuing read requests. Or, you can interpret this character as having some other purpose within your system. Control-Z is not recognized by the minimal Terminal Handler.

Line-Edit Controls.

RUBOUT. RUBOUT deletes the last character from the line buffer and echoes the deleted character to the display. A BEL character is echoed if the operator attempts to RUBOUT past the beginning of the logical line.

Control-R. Control-R issues a CR,LF sequence and then echoes a copy of all characters entered since the last break sequence. The control-R character itself has no echo. Control-R is useful in generating a clean copy of the current line, minus all characters you have deleted using RUBOUT.

Control-X. Control-X deletes all characters entered into the line buffer since the last break sequence; that is, it deletes the current input line. Control-X is useful for correcting errors that are inconvenient to correct using RUBOUT. Control-X echoes a pound-sign (#) after the characters to be deleted and then issues a Carriage Return and a Line Feed.

Control-P. Control-P causes the next character entered to be interpreted as data. This control is useful when you want to enter one of the control characters as a part of an input line. For example, you might want to enter a partially formatted line that is to be written out later with a Carriage Return and a Line Feed. A control-P must precede each of these control characters, or they will perform their normal functions. Control-P is not recognized by the minimal Terminal Handler.

Output Controls.

Control-S. Control-S suspends the display of output through the RQOUTX exchange. It does not affect the RQALRM exchange; consequently, Debugger output continues if the Debugger is active. Tasks can continue producing output data while control-S is in effect, but the data cannot be displayed until a control-Q is issued. Control-S has no echo. It is not recognized by the minimal Terminal Handler.

Control-Q. Control-Q resumes output through the RQOUTX exchange, thus negating the effect of the control-S. Control-S and control-Q are especially useful for suspending output during a debugging session. Notice that leaving a control-S in effect indefinitely may eventually delay execution of tasks that use RQOUTX for output when all their output buffers are filled. Control-Q has no echo. It is not recognized by the minimal Terminal Handler.

Control-O. Control-O kills all output through the RQOUTX exchange. Control-O differs from control-S in that data that is normally written out through the RQOUTX exchange is simply ignored and therefore lost. Control-O remains in effect until a subsequent control-O is encountered. Control-O has no echo on the full Terminal Handler; on the minimal Terminal Handler an echo of O is provided.

Control-C. As mentioned previously, control-C is a special control used to invoke the Debugger. Its exact function is to send a message of type CNTRLCTYPE to the RQWAKE exchange; it also clears the Terminal Handler's type-ahead buffer.

Control Table.

The control table RQCTAB is a twelve-byte positional table, used by the full Terminal Handler to recognize control command characters. The full Terminal Handler provides a default control table with the values shown in table 4-2.

Table 4-2. Full Terminal Handler Control Characters

POSITION	CHARACTER	VALUE
0	CR	0DH
1	LF	0AH
2	Control-S	13H
3	Control-Q	11H
4	Control-O	0FH
5	Control-R	12H
6	ESC	1BH
7	RUBOUT	7FH
8	Control-C	03H
9	Control-X	18H
10	Control-Z	1AH
11	Control-P	10H

You may specify a substitute value for any control command. You may also cancel the effect of a control by assigning it a value greater than or equal to 80H.

The predefined RQCTAB shown above is contained in the Terminal Handler's THI820.LIB, THI830.LIB, or THI810.LIB, depending on whether the iSBC 80/20, 80/30, or 80/10 version is used. You can replace RQCTAB by deleting it from the appropriate library and then inserting your own version of RQCTAB in the same library. If you want to supply a substitute RQCTAB and still save the system version of the table, you can build RQCTAB in your configuration module and declare it PUBLIC. The LINK program will then link in only the first version as long as you link in your configuration module before the Terminal Handler libraries.

Operator Summary Sheet. When you design a system that includes the Terminal Handler, you must be certain that an operator is able to use your system. Perhaps the simplest way to do this is to provide a summary sheet for the operator. The following information can be used as a basis for such a summary sheet. (Note that some details will vary depending upon the individual characteristics of your terminal.)

This sample operator summary sheet covers the capabilities of the full input-output Terminal Handler. If you are using the full output-only version or either of the two minimal Terminal Handlers, your summary sheet should not cover some of these features.

System Start-Up

1. Turn ON the terminal.
2. Make certain that the terminal is ON-LINE, not local.
3. Turn ON the system.
4. If necessary, strike the U (uppercase U) until the letters appears on the display.

Editing Rules

To End Input Line: Strike the Carriage Return, Line Feed, or the ESCAPE key.

To Delete a Typo: Strike the RUBOUT key to delete the previous character. If you do this repeatedly, the display will begin to form a mirror image of the information already entered (CAT becomes CATTAC). When you reach the letter to be corrected, simply retype the correct information.

To Delete an Entire Line: Strike the Control and X keys together. The deleted line remains on your display, but is not entered into the system. A # character is placed at the end of the line to indicate that it has been deleted. You can then reenter the correct data.

Output Controls

Sometimes you need to enter data at the same time the system is writing data on your terminal. The following controls halt normal output. Alarms will still appear on your terminal.

To Suspend Output: Strike the control and S keys together. If you leave the output suspended too long, system performance may be affected when too much data is waiting for output.

To Resume Output: Strike the control and Q keys together.

To Kill Output: Strike the control and O (letter O) keys together. This control causes the loss of output data. To resume output, strike the control and O keys together again. This restarts the output, but all data that would have been written between the two commands is lost.

End Of Input Control

To End Input: Strike the control and Z keys together. This tells the system that you have finished entering data and no longer require the terminal. (Note to programmer: This will only be true if your program interprets control-Z as end-of-input, as discussed previously under "Line-Break Controls.")

Configuration, Linking, and Locating

This section supplies the information you need to include the Terminal Handler in your configuration module, and to link and locate the resulting object code. Instructions for preparing the configuration module from this information are given in Chapter 3, along with general instructions for linking and locating your system.

Configuration Requirements

Output-Only Version.

Tasks. One task must be defined, RQTHDO. The stack length of this task is 46 bytes; priority must be 127; and the default exchange is RQOUTX.

Initial Exchanges. You must declare the following exchanges external:
- RQOUTX
- RQALRM (full Terminal Handler only)
- RQL7EX (interrupt exchange)
- One user-defined exchange. This exchange is required to receive the write request message response from the Terminal Handler.

PUBLIC Variables. You must supply a value for RQRATE and declare it PUBLIC, unless the required value is the UNRSLV.LIB default. Note that the default cannot be used for the output-only version, since this value specifies the automatic baud rate search, which is only available with the input-output version. (Refer to "Terminal Baud Rate" earlier in this chapter.)

Input-Output Version.

Tasks. One task must be defined, RQTHDI. The stack length of this task is 36 bytes; priority must be 112; and the default exchange is RQOUTX.

Initial Exchanges. You must declare the following exchanges PUBLIC:
- One user-defined exchange to receive read request message responses from the Terminal Handler.
- One user-defined exchange to receive write request message responses from the Terminal Handler.

You must declare the following exchanges EXTERNAL:

- RQINPX
- RQOUTX
- RQWAKE
- RQDBUG (full Terminal Handler only)
- RQALRM (full Terminal Handler only)
- RQL7EX (interrupt exchange)
- RQL6EX (interrupt exchange)

NOTE

If a single task performs all input and output through the Terminal Handler, it may be possible to use one user-defined exchange as a response exchange for both functions. When many tasks perform input and output, it may be necessary for each task to have its own user-defined exchange(s) to receive responses from the Terminal Handler.

PUBLIC Data. For the full Terminal Handler, you must supply a value for RQRATE and declare it PUBLIC, unless the required value is the UNRSLV.LIB default. (Refer to "Terminal Baud Rate" earlier in this chapter.) For the minimal Terminal Handler, RQRATE must be declared.

Linking and Locating

The following files must be added to the list of files given to the LINK program to link the full Terminal Handler into your system:

1. THI820.LIB, THI830.LIB, or THI810.LIB (for the iSBC 80/20, 80/30, or 80/10 version, respectively; required only for the input-output Terminal Handler)

2. THO820.LIB, THO830.LIB, or THO810.LIB (for the iSBC 80/20, 80/30, or 80/10 version, respectively)

For linking the minimal Terminal Handler, the files to be added are the following:

1. MTI820.LIB, MTI830.LIB, or MTI810.LIB (for the iSBC 80/20, 80/30, or 80/10 version, respectively; required only for the input-output Terminal Handler)

2. MTO820.LIB, MTO830.LIB, or MTO810.LIB (for the iSBC 80/20, 80/30, or 80/10 version, respectively)

Whether your Terminal Handler is full or minimal, both files must be given in the order shown for the input-output version. The first file may be omitted for the output-only version. However, no additional code is linked into the output-only version when both file names are given.

No special information is needed to locate object code for a system that includes the Terminal Handler. For general instructions, refer to Chapter 3.

Table 4-3. Summary of Features for Configurations of the Terminal Handler

Feature	Full Terminal Handler		Minimal Terminal Handler	
	Input-Output	Output-only	Input-Output	Output-only
Exchanges:				
RQINPX	YES	NO	YES	NO
RQOUTX	YES	YES	YES	YES
RQWAKE	YES	YES	YES	YES
RQALRM	YES	YES	NO	NO
RQDBUG	YES	NO	NO	NO
RQL6EX	YES	NO	YES	NO
RQL7EX	YES	YES	YES	YES
Control Commands:				
CR	YES	NO	YES	NO
LF	YES	NO	NO	NO
ESC	YES	NO	NO	NO
Control-Z	YES	NO	NO	NO
RUBOUT	YES	NO	YES	NO
Control-R	YES	NO	YES	NO
Control-X	YES	NO	YES	NO
Control-P	YES	NO	NO	NO
Control-S	YES	NO	NO	NO
Control-Q	YES	NO	NO	NO
Control-O	YES	NO	YES	NO
Control-C	YES	NO	YES	NO
Other:				
Control table (RQCTAB)	YES	NO	NO	NO
Type-ahead buffer	YES	NO	NO	NO

General Description

Random-access memory (RAM) is one of the most valuable resources in a computer system. In many applications, however, RAM is used so ineffectively that it is almost wasted. Assume, for example, that an initialization task executes only once and then is left unused until the next time the system is powered up. The RAM required for the initialization task's messages is also unused most of the time. The Free Space Manager is provided to help you make the most efficient use of RAM in your system, by allowing you to reclaim RAM that is no longer needed by a task and make it available for use by other tasks. Thus the Free Space Manager allows you to treat RAM as a common resource to be shared dynamically by a number of tasks.

Dynamic sharing of RAM via the Free Space Manager can substantially reduce the amount of RAM required in a system. As an example, assume that five tasks each require 2K of RAM, or a total of 10K. If these tasks are not normally in contention for system resources, it may be possible to reduce the RAM requirements for the tasks to 2K. Rather than permanently assign 2K blocks of RAM to each task, you can assign a single 2K block to the Free Space Manager. When a task needs to build a message or needs a buffer for some other purpose, the task requests the desired amount of RAM from the Free Space Manager. When the RAM is no longer needed, the task can surrender it back to the Free Space Manager so that it can be reused.

The Free Space Manager is especially useful when a system deals with variable-length messages. Without the Free Space Manager, you must allocate enough RAM to accommodate the longest possible message. Many systems use maximum-length messages only infrequently; therefore, the system does not get the maximum possible use from its RAM allocations. With the Free Space Manager, variable-length messages can be allocated from a common pool of RAM. When each message is assigned the amount of RAM required, any remaining RAM is available for other uses.

Use Environment

A single version of the Free Space Manager operates on the iSBC 80/20, 80/30, and 80/10. (Note, however, that the object code files for the Free Space Manager on the iSBC 80/20, 80/30, and 80/10 Executive diskettes have different names. Refer to Appendix A.)

Memory and Hardware Requirements

The Free Space Manager requires less than 950 bytes of code space and approximately 200 bytes of data space; it uses a 40-byte stack. Refer to Appendix E for exact byte counts.

No special hardware requirements are imposed by the Free Space Manager.

How the Free Space Manager Operates

The Free Space Manager provides two basic functions:

- It maintains a pool of free RAM and, upon request from a task, allocates space from the pool.

- It can accept blocks of RAM that are no longer needed in the system and return them to the pool.

The Free Space Manager treats its blocks of RAM as messages; in fact, it uses certain message fields in the first eight bytes of the block, as described later in this section. However, you are free to use the RAM, including the message heading portion of the block, for any purpose once it has been allocated.

Figure 5-1 illustrates the relationships between a user task and the Free Space Manager. (Though shown here as a single box, the Free Space Manager is actually a collection of tasks.) A user calls on the Free Space Manager by sending a message to one of two Free Space Manager exchanges. To request a RAM allocation, a task sends a message to the Free Space Manager's allocation exchange, RQFSAX. The Free Space Manager responds by returning the same message to a user-defined response exchange (RESP$EX). If the allocation has been made, the returned message contains the address of a block of RAM of the desired size. If the allocation has not been made, the message contains the length of the largest currently available block of RAM.

To surrender a block of RAM, the task sends a message to the Free Space Manager's reclamation exchange, RQFSRX. The task surrendering RAM to the Free Space Manager does not receive a response message.
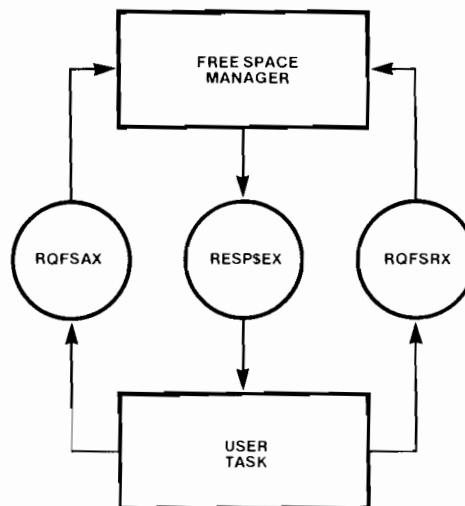
Figure 5-1. Free Space Manager Exchanges

The size of an allocated block of RAM is controlled by two rules. First, the smallest block available is eight bytes. Second, blocks are always allocated in four-byte increments. Should a task request a block which does not satisfy both rules, the Free Space Manager will attempt to allocate a slightly larger block that does conform to the rules.

Using the Free Space Manager

Initialization

Before any allocation requests are made, a user task must initialize the Free Space Manager by telling it what areas of RAM are available for allocation. This is accomplished by sending one or more messages to the reclamation exchange of the Free Space Manager (RQFSRX). The Free Space Manager sends no response.

Each message sent to the reclamation exchange represents a block of RAM to be made availabe for allocation. Each message must be physically located within the block that it represents. Before each message is sent to the Free Space Manager, your task must initialize the LENGTH field (bytes 2 and 3 of the block). The number stored in this field must be the number of bytes in the entire block. If this number is

not a multiple of four, the Free Space Manager will reduce the block size to the largest multiple of four. If the LENGTH field is set to less than four, the Free Space Manager will not reclaim the block.

To reduce system overhead, do not send RAM that cannot possibly be allocated by the Free Space Manager. If, for example, the smallest allocation request in a system is for twenty bytes, you should not assign non-contiguous blocks of less than twenty bytes to the Free Space Manager. To do so requires the Free Space Manager to keep track of this unused memory. This caution does not apply to contiguous blocks of memory. When the Free Space Manager encounters separate but contiguous blocks of free RAM, it merges the separate blocks together to form the largest possible single block. Therefore, if you initialize the Free Space Manager by sending it three 100-byte blocks of RAM and the blocks happen to be contiguous, the Free Space Manager treats the memory as a single 300-byte block.

Allocation Request

Allocation Request Message. A task requests a memory allocation by sending a message to the Free Space Manager's allocaton exchange, RQFSAX. The format for a memory allocation request message is shown in figure 5-2. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those that may be changed by the Free Space Manager when it returns the message.

0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	MSG LENGTH*

Figure 5-2. Allocation Request Message

Those fields that have specific meaning in the allocation request message are described in the following paragraphs. (For general information, refer to "Message Coding" in Chapter 2.)

LENGTH is 11. (Note that this is the length of the *allocation request message,* not the length of the desired memory allocation.)

TYPE must be either FSREQTYPE (4), for a conditional request, or UCREQTYPE (5), for an unconditional request. These types are explained in the paragraphs that follow these message field definitions.

HOME EXCHANGE is not used by the Free Space Manager.

RESPONSE EXCHANGE must specify the address of the user-defined exchange where the requesting task waits for a response from the Free Space Manager.

MSG LENGTH must specify the length of bytes of the block of RAM to be allocated. The value should be a multiple of four bytes, and must be at least eight bytes. The Free Space Manager rounds up to the nearest multiple of four any request that is not a multiple of four.

Conditional Request. When an allocation request of type FSREQTYPE (4) is sent, the Free Space Manager responds by returning the user's request message with the TYPE field set to indicate either a positive or a negative acknowledgment.

The TYPE field remains unchanged when a positive acknowledgement is returned, and the MSG LENGTH field is set to the *address of the allocated block of RAM.*

For a negative acknowledgment, the TYPE field is set to FSNAKTYPE (6), and the MSG LENGTH field indicates the *size of the largest available block of contiguous RAM*. The requesting task must test the TYPE field to determine whether the allocation was made, then take appropriate action.

For a positive acknowledgement, the Free Space Manager also supplies information in certain fields of the allocated block of RAM, as described later under "Format of Allocated Message."

If the acknowledgment is negative, no RAM is allocated. The requesting task can check the MSG LENGTH field to determine the size of the largest available RAM block; if the task can use a block of that size, it can issue a second request message. Note, however, that the amount of available RAM is subject to change if some other task requests or surrenders RAM before the second request is honored.

In making conditional requests, care must be taken to avoid a "lockout" situation. If a task loops continuously, making a conditional allocation request on each iteration until it gets a positive acknowledgment, but never waiting at an exchange, and if this task has a relatively high priority, it will "lock out" the Free Space Manager reclamation and merge tasks, so that no memory will ever be freed to satisfy the requesting task. This situation can occur with a Disk File System READ or DISKIO request if dynamic buffer allocation is selected (see Chapter 7, "Buffers"), since DFS makes a conditional allocation request to the Free Space Manager. The situation can be avoided by having the requesting user task perform a timed RQWAIT operation (at an exchange where no messages are sent) each time around the loop, giving other tasks the opportunity to run.

Unconditional Request. You can avoid the logic needed for testing the response type by issuing an allocation request of type UCREQTYPE (5). This type of message never produces a negative acknowledgment. When an unconditional request is issued, the Free Space Manager responds with a positive acknowledgment (as described under "Conditional Request") if the request can be filled. However, if the request cannot be filled, the requesting task waits indefinitely until sufficient memory becomes available. At that time, a positive acknowledgment is returned.

When the RQWAIT operation is used to wait for a response to an unconditional request, the RQWAIT should not normally specify a time limit. To do so may allow your task to time out and resume execution before its allocation request has been honored. However, the unconditional request remains in effect and will eventually be filled. Since the task is no longer waiting for a response, it may be unaware that the request has been filled. This situation creates many possibilities for problems and should be avoided.

Format of Allocated Message. Whenever the Free Space Manager is able to grant a request for a block of RAM, two messages are generated. The first message, called the allocated message, is built within the block of allocated RAM. Its fields (illustrated in figure 5-3) are described in the following paragraphs. The second message generated is a modification of the request message that was sent to the Free Space Manager. It differs from the request message only in one regard—the MSG LENGTH field of the message now contains the address of the allocated block of RAM. It is this second message that is sent to the response exchange field designated in the request message.

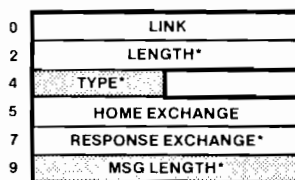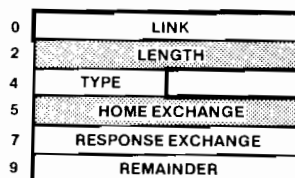**Figure 5-3. Allocated Message**

Those fields that have specific meaning in the allocated message are described in the following paragraphs. (For general information, refer to "Message Coding" in Chapter 2.)

LENGTH is set equal to MSG LENGTH from the allocation request message (rounded up to a multiple of four). This indicates the total length of the allocated message (block).

TYPE is not used by the Free Space Manager.

HOME EXCHANGE is set to the address of the Free Space Manager's reclamation exchange, RQFSRX.

RESPONSE EXCHANGE is not used by the Free Space Manager. (In an eight-byte allocation, only one byte of this field is present.)

The REMAINDER is an undefined user area.

The address returned in the MSG LENGTH field of the response message provides access to the allocated RAM. PL/M programmers can use this as a base address. Assembly language programmers can load this address into the H and L registers and use it as a base for addressing the allocated RAM.

Although the RAM is allocated as a message, you need not use it as such. You might, for example, use the RAM as a scratchpad for a calculation. However, if you write over the LENGTH field of the message heading, you must restore the LENGTH before sending the RAM back to the Free Space Manager for reclamation.

Timing Considerations. When a task requests a RAM allocation, the requesting task typically waits while the Free Space Manager attempts to allocate the desired amount of contiguous memory. However, when the task waits, all ready tasks with priorities higher than the Free Space Manager execute before the Free Space Manager. Although the allocation routine itself is highly efficient, the possible existence of higher-priority ready tasks makes it difficult to predict the elapsed time between a RAM request and the actual allocation. There are two potential solutions to this problem:

* The priority of the Free Space Manager is user-assigned when the system is configured. This allows you to assign the same priority to both the Free Space Manager and a task with a critical timing requirement. This will insure that only tasks with priorities higher than that of the requesting task will preempt the Free Space Manager.

* The task can request the RAM allocation before it reaches the code with critical timing requirements.

Remember that the Free Space Manager treats RAM as a shared resource. When many tasks contend for this resource, the Free Space Manager may temporarily be unable to fulfill a request. This may cause a task making an unconditional request to wait until enough RAM is returned to the pool to fill the request.

Allocation Technique. The Free Space Manager maintains a list of available blocks of RAM. This list indicates the starting address and length of each block. However, to make allocations as quickly as possible, the Free Space Manager allocates memory on a "first-fit" rather than a "best-fit" basis. For example, assume that blocks of 40, 28, and 100 bytes are available when a request for a 28-byte block is honored. The 28-byte allocation is made from the 40-byte block, thus leaving a 12-byte fragment. This 12-byte fragment may or may not be used if there is a subsequent request for an eight- or twelve-byte allocation.

Reclamation

Sending a Message for Reclamation. A task returns a block of RAM to the Free Space Manager by sending it in the form of a message to the Free Space Manager's

reclamation exchange, RQFSRX. If the LENGTH field (the third and fourth bytes) of the block has been changed since the block was allocated, or if the length of the block being returned is different, you must set this field. The length must be that of the memory block being returned, and should be a multiple of four bytes. The Free Space Manager rounds down any LENGTH fields that are not multiples of four. This is the only field in the message that need be set before it is sent.

When a user task receives an allocation of memory from the Free Space Manager, the task should save the address of the allocated message (from the MSG LENGTH field of the response message). When the allocated message is no longer needed, the task need only set the LENGTH field of the message (if it has been changed), then call RQSEND with the address of RQFSRX and the message address as parameters.

Reclamation Technique. When a block of RAM is surrendered, the Free Space Manager checks to see if it is adjacent to any other block of available RAM. If so, the Free Space Manager merges the blocks into a single, larger block. The task that performs these merges runs at priority level 254. The task only uses processor time that would otherwise be spent in a HALT state.

Note that when many tasks are making heavy use of system resources, the merge task may not be able to run often enough to satisfy the needs of the system. To alleviate this problem you can: a) avoid returning memory to the Free Space Manager when a task needs to use it again in a relatively short time, thus lightening the load on the Free Space Manager; or b) provide the Free Space Manager with more memory, so that spare memory will almost always be available.

NOTE

When the Free Space Manager accepts a block of memory from one of your tasks, it does not check for overlap. If your task frees a block of RAM and claims that it is longer than it actually is, the Free Space Manager will accept the incorrect size. Errors of this nature can result in a single block of RAM being allocated to two different tasks simultaneously.

Coding Examples

The basic steps for requesting a memory allocation are:

- Build a request message
- Send (RQSEND) the message to the Free Space Manager
- Wait (RQWAIT) for a response from the Free Space Manager
- If the request was conditional, check TYPE field of response message to ensure that the allocation was successful.

The basic steps in returning memory to the Free Space Manager are:

- Set the LENGTH field (third and fourth bytes) of the block of memory being returned
- Send (RQSEND) the memory as a message to the Free Space Manager.

PL/M. In the following PL/M code, assume that EXTERNAL declarations have been provided for the Free Space Manager exchanges RQFSAX and RQFSRX, that the exchange REQEX has been created and declared for use in communication with the Free Space Manager, and that the address variable ALLOCMSGADDR has been declared. Also assume that the Free Space Manager has been initialized by sending one or more blocks of memory, in the form of messages, to RQFSRX.

The following code builds a memory allocation request message:

```
/* DECLARE ALLOCATION REQUEST MESSAGE */
DECLARE                        REQMES                           STRUCTURE (
                               LINK                             ADDRESS,
                               LENGTH                           ADDRESS,
                               TYPE                             BYTE,
                               HOME$EXCHANGE                    ADDRESS,
                               RESPONSE$EXCHANGE                ADDRESS,
                               MSG$LENGTH                       ADDRESS);

       /* BUILD ALLOCATION REQUEST MESSAGE */

       REQMES.LENGTH = 11;
       REQMES.TYPE = FS$REQ$TYPE;
       REQMES.RESPONSE$EXCHANGE = .REQEX;
       REQMES.MSG$LENGTH = 40H;
```

After the message is built, the user task sends it to the allocation exchange and waits for a response:

```
       CALL RQSEND(.RQFSAX,.REQMES);
       SCRATCH = RQWAIT(.REQEX,0);
```

The task checks the TYPE code returned by the Free Space Manager to see if the request was honored. If not (TYPE=FSNAKTYPE), the task requests allocation repeatedly until the allocation is obtained. Note that in all requests for allocation after the first, the task voluntarily surrenders the processor by waiting for one system time unit. (This example assumes that no message will arrive at the REQEX exchange during the wait.) This gives other application tasks and the Free Space Manager reclamation and merge tasks a chance to run and to enlarge the RAM pool. Other tasks contending for the system, however, could still prevent the reclamation and merge tasks from gaining control:

```
       DO WHILE REQMES.TYPE = FS$NAK$TYPE;
                       SCRATCH = RQWAIT(.REQEX, 1);
                       REQMES.TYPE = FS$REQ$TYPE;       /*RESET TYPE*/
                       REQMES.MSG$LENGTH = 40H;         */RESET LENGTH*/
                       CALL RQSEND(.RQFSAX,.REQMES);
                       SCRATCH = RQWAIT(.REQEX,0);
                       END;

       /*SAVE ADDRESS OF ALLOCATED BLOCK*/

       ALLOC$MSG$ADDR = REQMES.MSG$LENGTH;
```

This example does not try to use a smaller allocation when it receives a negative acknowledgment (MSG$LENGTH contains the size of the largest block currently available). Nor does it make use of the time available to it to perform other processing between successive requests for allocation. Therefore it would be more appropriate for the task to make an unconditional request and to wait indefinitely until the allocation can be made:

```
       /*BUILD UNCONDITIONAL REQUEST MESSAGE*/

       REQMES.LENGTH = 11;
       REQMES.TYPE = UC$REQ$TYPE;
       REQMES.RESPONSE$EXCHANGE = .REQEX;
       REQMES.MSG$LENGTH = 40H;
```

```
/*REQUEST ALLOCATION & WAIT UNTIL OBTAINED*/

CALL RQSEND(.RQFSAX,.REQMES);
SCRATCH = RQWAIT(.REQEX,0);

/*SAVE ADDRESS OF ALLOCATED BLOCK*/

ALLOC$MSG$ADDR = REQMES.MSG$LENGTH;
```

Assuming that the LENGTH field of the allocated block has not been altered, the block can be returned to the RAM pool by sending it to the reclamation exchange:

```
CALL RQSEND(.RQFSRX,ALLOC$MSG$ADDR);
```

Assembly Language. The following assembly language code performs the same functions as the PL/M code given in the preceding section. Assume that RQFSAX and RQFSRX have been declared external, that the user exchange REQEX has been created and declared, and that REQMES has been declared in dedicated RAM.

```
AGAIN: LXI    H,11H        ;LOAD H&L WITH MESSAGE LENGTH (11)
       SHLD   REQMES+2     ;SET MESSAGE LENGTH
       MVI    A,4H         ;LOAD ACC WITH MESSAGE TYPE (4)
       STA    REQMES+4     ;SET MESSAGE TYPE
       LXI    H,REQEX      ;LOAD H&L WITH ADDRESS OF RESPONSE EXCHANGE
       SHLD   REQMES+7     ;SET RESPONSE EXCHANGE ADDRESS
       LXI    H,40H        ;LOAD H&L WITH MSG LENGTH (40H)
       SHLD   REQMES+9     ;SET MSG LENGTH TO 40H
```

Before calling the RQSEND operation, the program must load the exchange address and message address into BC and DE register pairs, respectively:

```
       LXI    B,RQFSAX     ;LOAD B WITH ALLOC EXCHANGE ADDRESS
       LXI    D,REQMES     ;LOAD D WITH REQUEST MESSAGE ADDRESS
       CALL   RQSEND
```

The task now waits at the REQEX exchange until the Free Space Manager responds. First, however, the B and D registers must be loaded with the address of the REQEX exchange and the RQWAIT time limit:

```
       LXI    B,REQEX      ;LOAD B WITH RESPONSE EXCHANGE
       LXI    D,0H         ;NO SPECIFIC TIME LIMIT FOR THIS WAIT
       CALL   RQWAIT
```

The TYPE code returned by the Free Space Manager is tested and the request is re-issued until it is satisfied. The task waits for one system time unit between requests:

```
       LXI    H,REQMES+4   ;H&L TO POINT TO TYPE FIELD
       MVI    A,4H         ;POSITIVE ACK IN ACCUMULATOR
       CMP    M            ;POSITIVE ACK IN TYPE?
       JZ     GOTIT        ;YES, CONTINUE
       LXI    B,REQEX      ;NO, SURRENDER FOR 1 STU
       LXI    D,1H
       CALL   RQWAIT
       JMP    AGAIN        ;TRY AGAIN

GOTIT: EQU    O
       LHLD   H,REQMES+9   ;GET ADDRESS OF BLOCK
       PUSH   H            ;SAVE ON STACK
```

At this point the program has copies of the message address in the H and L registers and on the stack. The program can gain access to any location in the message by incrementing the message address in the H and L registers.

As soon as the message is no longer required, it is returned to the Free Space Manager.

```
LXI    B,RQFSRX      ;LOAD RECLAMATION EXCHANGE ADDRESS
POP    D             ;LOAD D WITH MESSAGE ADDRESS
CALL   RQSEND        ;SEND MESSAGE TO FREE SPACE MANAGER
```

Configuration, Linking, and Locating

This section supplies the information you need to include the Free Space Manager in your configuration module, and to link and locate the resulting object code. Instructions for preparing the configuration module from this information are given in Chapter 3, along with general instructions for linking and locating your system.

Configuration Requirements

Tasks. One task must be defined, RQFMGR. The stack length of this task must be 40 bytes, and the default exchange is RQFSRX.

The Free Space Manager may be assigned any priority level that is appropriate within your system as mentioned previously. Assigning the Free Space Manager the same priority assigned to a task that uses its services ensures the Free Space Manager that it will not have to compete for the processor with lower-priority tasks while it services the request. If none of the tasks that use the Free Space Manager have critical timing requirements, the Free Space Manager can be assigned the lowest priority in the system; it will then execute only when no other task is ready, and therefore uses only processor time that would otherwise be idle. However, if the system is heavily loaded, the tasks that use the Free Space Manager may be forced to wait indefinitely for their allocation requests to be serviced.

Initial Exchanges. You must declare two exchanges external: RQFSAX and RQFSRX. Also, at some point you must create a response exchange. This can be done either dynamically or at configuration time.

PUBLIC Data. No PUBLIC data items need to be defined at configuration time for the Free Space Manager.

Linking and Locating

To link the Free Space Manager into your system, one file must be added to the LINK list. This file is TSK820.LIB for the iSBC 80/20 version, TSK830.LIB for the iSBC 80/30, or TSK810.LIB for the iSBC 80/10.

No special information is needed to locate object code for a system that includes the Free Space Manager; for general instructions, refer to Chapter 3.

General Description

RMX/80 provides a Debugger especially designed for use in the RMX/80 real-time environment. Because the real-time environment presents certain unique debugging situations, this Debugger is provided in addition to other Intel diagnostic tools such as the ICE-80 or ICE-85 In-Circuit Emulator. The Debugger can be used in conjunction with ICE-80 or ICE-85 to obtain some capabilities (such as memory mapping and debugging an inoperative system) not available in the Debugger alone.

The Debugger allows you to communicate with your RMX/80 system through the Terminal Handler. The Debugger offers services beyond those provided by ICE-80 and ICE-85. For example, the Debugger can format and display RMX/80 structures such as Task Descriptors and Exchange Descriptors. Tracing the dynamic changes in these structures can give you a good understanding of what is happening in the system. Used in this fashion, the Debugger can be a powerful educational tool, and can also help improve the performance of systems that are executing correctly.

A basic understanding of RMX/80 system operation will aid you considerably in debugging your system. The "Details of System Operation" section of Chapter 2 supplies helpful information on this subject. In using the Debugger, you will probably want to refer often to this section, particularly the system control structures diagram at the end of the chapter (figure 2-8).

Debugger Capabilities

The RMX/80 Debugger is provided in two versions, an active Debugger and a passive Debugger. The active Debugger allows you to look into the system and to modify it. The passive Debugger allows you to look into the system, but does not allow you to modify it.

Because it offers a greater number of features, the active Debugger is the logical choice for debugging a prototype system with sufficient memory or when you are using ICE-80 or ICE-85 with an Intellec Development System. Because it requires less memory, the passive Debugger may be the only choice when memory is limited and when ICE-80 or ICE-85 is not available.

The passive Debugger allows you to:
- Examine the contents of specified locations in memory.
- View the contents of system lists, including the Task List, Exchange List, Ready List, Suspend List, Delay List, and the list of all tasks or messages waiting at any specified exchange.
- Look at formatted system control structures, including Task Descriptors, Exchange Descriptors, and messages.
- Quit the debugging session.

The active Debugger includes all the features of the passive Debugger, and also allows you to:
- Modify the contents of specified memory locations and of fields in system lists, Task Descriptors, Exchange Descriptors, and messages.
- Define symbolic names for numeric values such as addresses (and change or remove the names if necessary).

- Execute an RMX/80 or user-supplied procedure.

- Set breakpoints at memory locations (execution breakpoints) or on exchanges (sends or waits).

- Display and/or modify the registers of a task that has incurred a breakpoint.

- Set up a periodic scan of task stacks for overflow conditions.

One special feature of the RMX/80 Debugger is that the system does not stop when a breakpoint occurs; the task incurring the breakpoint is (optionally) suspended, but the rest of the system continues to run. This feature is especially important in the real-time environment, when it is crucial that certain processes not be interrupted.

Use Environment

The Debugger differs from other RMX/80 extensions in that it is more likely to be used for testing in a prototype system than in a final application system. There is relatively little need for the Debugger in an already thoroughly debugged system. However, some users may elect to include the Debugger in systems that may require field troubleshooting.

The Debugger requires the services of the full input/output Terminal Handler, which includes a number of exchanges and controls that support the Debugger. (The system being debugged must include a terminal.)

The Debugger is designed for use with the iSBC 80/20, 80/30, and 80/10 Single Board Computers. Because of differences in interrupt handling, there is one version of the Debugger for the iSBC 80/20 and 80/30, and another version for the iSBC 80/10.

Memory and Hardware Requirements

The active Debugger requires less than 11K (K=1024) bytes of code space and less than 1700 bytes of data space. The passive Debugger requires less than 4K bytes of code space and less than 600 bytes of data space. Both active and passive Debuggers use a 64-byte stack. Refer to Appendix D for exact byte counts.

No special hardware requirements are imposed by the Debugger in a fully operating system, other than the Terminal Handler requirements (see Chapter 4). However, refer to the following section, "Using the Debugger," for the minimal hardware configuration necessary to run the Debugger during system development.

How the Debugger Operates

The Debugger operates in close conjunction with the Terminal Handler. The sections in Chapter 4 entitled "Terminal Handler Exchanges" and "Debugger Exchanges" describe the relationships between the Terminal Handler, the Debugger, user tasks, and exchanges. These relationships are summarized in figure 6-1.
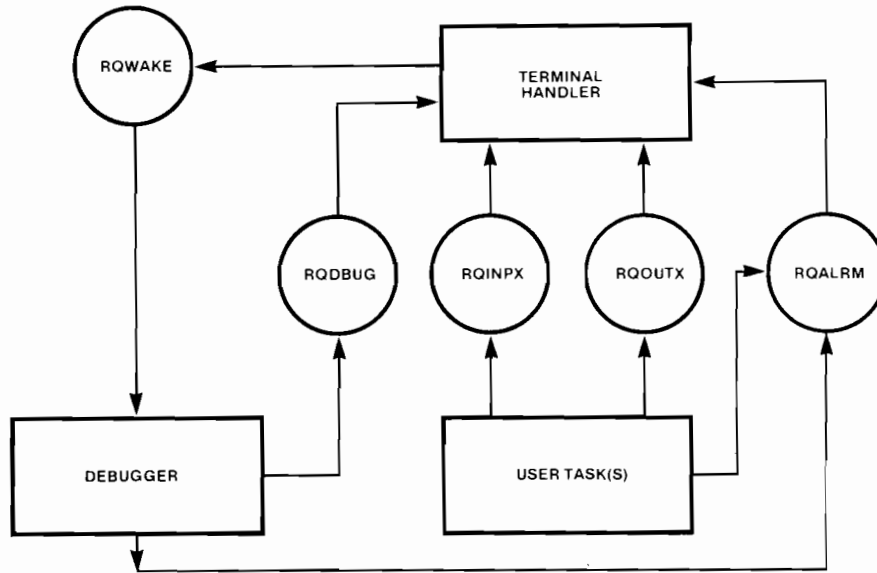
Figure 6-1. Terminal Handler-Debugger Relationships

Using the Debugger

Before using the Debugger in a developing system, you must decide how you wish to map Debugger code into your prototype memory, and must also establish an operational hardware configuration. These considerations are discussed in this section.

Once the hardware and software are configured for your prototype system (refer to Chapter 3 for software configuration), you communicate with the Debugger by a set of commands entered from the terminal. (The Terminal Handler must be included in your prototype system.) Following the discussions of memory mapping and hardware configuration, this chapter provides a complete description of all Debugger commands, a list of error messages, and some general debugging suggestions.

Memory Mapping of Debugger Code

For most RMX/80 applications, the final goal is to have the code be ROM-resident. The Debugger, however, is primarily useful during system development, and may not be included at all in the final ROM-resident application system. Thus it is useful

here to mention the different memory mapping options that may be chosen. Debugger code may be mapped into a system in several ways:

*   Debugger code may be placed in your system's PROM.

*   Debugger code may be placed in your system's RAM via ICE-80 or ICE-85.

*   Debugger code may be loaded into Intellec Microcomputer Development System memory by using the ICE-80 or ICE-85 In-Circuit Emulator.

The first two options require your system to include enough memory to accommodate the Debugger. Typically, this implies the use of a memory expansion board.

Because of the cost of programmable ROM's, Debugger code is normally placed in ROM only when it is to be part of the application system. This option is useful for field-testing systems.

Prototype systems with sufficient RAM sometimes locate Debugger code in RAM. Application systems typically avoid loading code into RAM, since this requires the use of a peripheral device for program loading.

The third option—using ICE-80 to map the Debugger into the Intellec Development System's memory—is the most practical for typical systems. Notice, for example, that the Debugger must run under the RMX/80 Nucleus and requires the use of the Terminal Handler. Therefore, when RMX/80 is not operative, neither is the Debugger. The ICE-80 and ICE-85 diagnostics may be helpful in this situation.

Hardware Configuration

Before attempting to use the Debugger, you must be certain that your hardware configuration is fully operational. The most efficient way to do this is to load and operate the Demonstration System with ICE-80 (for iSBC 80/20 and 80/10 systems) or ICE-85 (for iSBC 80/30 systems). For installation of ICE-80 or ICE-85 in the RMX/80 environment, refer to Appendix F.

The Demonstration System, described in Appendix I, exercises the Nucleus, Terminal Handler, Free Space Manager, and Debugger, plus a simple command line interpreter. By using the Demonstration System, you can be certain that both your hardware configuration and the RMX/80 software are operating properly.

Certain hardware modifications, described in Appendix F, must be made in order to operate RMX/80 software on your iSBC system. Also, you must be certain that the jumper-wired addresses of your iSBC memory are appropriate for your system. Again, the easiest way to check this is by implementing the Demonstration System.

NOTE

Configuring your system hardware is basically a one-time learning process. After you learn the steps needed to bring up your prototype system, you can apply the same steps to application systems without repeating the testing described here.

Debugger Commands

Entering Commands. You communicate with the Debugger by means of the Debugger commands. All Debugger commands must be entered through a terminal running under the RMX/80 Terminal Handler.

You invoke the Debugger by entering a control-C at the terminal. This awakens the
Debugger and directs subsequent terminal entries to the Debugger rather than to
your system. (Control-C also clears the Terminal Handler's type-ahead buffer.) The
Debugger remains in effect until you enter the letter Q followed by a carriage return
at the terminal. This signifies the QUIT command, which deactivates the Debugger.
Subsequent terminal entries are directed to your system rather than to the Debugger.
The Debugger remains inactive until another control-C is entered.

Throughout this chapter, the term debugging "session" refers to the period that
starts when you invoke the Debugger with a control-C and ends when you return
control to the Terminal Handler via the QUIT command. It is assumed that the
system continues to run and is not restarted.

While the Debugger is active, the Terminal Handler is also still active; all Terminal
Handler control commands are operative. You may, for instance, correct an im-
properly entered Debugger command with a series of RUBOUTs, then generate a
fresh copy of the line using control-R. During a debugging session, you may wish to
suspend terminal output by giving the control-S command, or to kill the output with
control-O. If you do so, you should remember to reverse the command with
control-Q or another control-O when you quit the debugging session.

The Debugger indicates that it is ready for a command by displaying an asterisk (*)
as a prompt character. You enter a Debugger command by typing its op-code on
your terminal. Many op-codes may be followed by optional characters to specify
particular functions. You terminate the command by striking the Carriage Return
key. Certain commands allow you to enter a large number of options in a single en-
try rather than enter the command repeatedly.

The first four commands shown in table 6-1 (MEMORY, VIEW, FORMAT, and
QUIT) are common to both the active and the passive Debugger. However, the ac-
tive Debugger provides more options for certain commands than the passive Debug-
ger. (Refer to the detailed command descriptions later in this section.) The rest of
the commands are available only in the active Debugger.

Table 6-1. Debugger Commands

COMMAND	OP-CODE	FUNCTION
MEMORY	M	The MEMORY command produces a hexadecimal display of the value at the current memory address. Options allow you to specify a one-byte or two-byte word display; to alter the current address; and to modify the value at the current address.
VIEW	V	The VIEW command displays the contents of a particular RMX/80 system list. For example, V R displays the current contents of the Ready List. A number of options allow you to specify the list to be displayed.
FORMAT	F	The FORMAT command formats and displays the contents of a Task Descriptor, Exchange Descriptor, or message.
QUIT	Q	The QUIT command terminates the debugging session. The Debugger surrenders control and waits for you to enter another Control-C. All subse-quent terminal input is directed to your system rather than to the Debugger.
EXECUTE*	X	The EXECUTE command allows you to enter the address of a PL/M procedure or assembly language subroutine to be executed along with any parameters required by the procedure. For example, you might activate a task by executing an RQSEND operation that sends a message to the ex-change where the task is waiting.

*Available only in the active Debugger.

Table 6-1. Debugger Commands (Cont'd.)

COMMAND	OP-CODE	FUNCTION
DEFINE*	D	The DEFINE command allows you to assign PL/M-like symbols (numeric variables) to frequently-used memory addresses or other numeric values.
REMOVE*	Z	The REMOVE command allows you to remove a previously-defined numeric variable from the system.
CHANGE*		No op-code is specified for the CHANGE command. Instead, you name the item to be changed, followed by an equals sign and the new value. The CHANGE command allows you to establish or change a breakpoint, define a Breakpoint Task, alter the Breakpoint Task's registers, or change the value of a numeric variable previously entered via a DEFINE command.
DISPLAY*		No op-code is specified for the DISPLAY command. Instead, you name the item to be displayed. The DISPLAY command displays numeric variables, breakpoint registers, and various items associated with the currently defined Breakpoint Task. If, for example, R is specified, the contents of the task's registers are displayed (accumulator, flags, etc.).
GO*	G	The GO command resumes the Breakpoint Task—i.e., makes it ready to run again. When the task resumes running, execution continues with the next instruction to be executed when the break occurred.
SCAN*	S	The SCAN command periodically checks the Task Descriptors of all tasks in the system to determine whether any task's stack has an overflow condition. If so, the SCAN command issues an overflow message that identifies the task with the overflow condition.

*Available only in the active Debugger.

Using the Detailed Command Descriptions. The first four commands described — MEMORY (M), VIEW (V), FORMAT (F), and QUIT (Q) — are common to both the active and the passive Debugger. However, the active Debugger allows certain options not supported by the passive Debugger. In the descriptions of these first four commands, options available only for the active Debugger are shaded. Users of the passive Debugger cannot use the shaded options. The next seven commands—beginning with the EXECUTE command—apply only to the active Debugger.

In the format definitions, braces enclose the different options (arranged in a vertical column) available for a command. Square brackets enclose optional fields in a command entry. An ellipsis (. . .) indicates that allowable options may be repeated any number of times. The term "expression" denotes a hexadecimal constant, a numeric variable, or a series of these added or subtracted. (Examples: 21A2, .B, .C + A1.)

Tokens (individual items or options within an entry) may be either run together or separated by blanks, provided that alphanumeric characters belonging to consecutive tokens are always separated by at least one blank. This restriction does not apply to punctuation (non-alphanumeric) characters. For instance, a MEMORY command specified as M=!W321A is not permitted, but M=!W 321A is valid.

Numeric parameters must be entered in hexadecimal. (No "H" suffix is required.) Likewise, the Debugger gives responses in hexadecimal.

MEMORY Command.

Function. The MEMORY command produces a hexadecimal display of the contents of the current memory location. Options allow you to specify a one-byte (byte) or two-byte (word) display; to alter the current address; and to modify the value at the current address.

Format.

$$
M \left\{ \begin{array}{l}
\text{!B} \\
\text{!W} \\
\text{expression} \\
\text{expression:expression} \\
@ \\
I \\
\text{no entry} \\
\text{=expression} \\
\text{=M expression} \\
\text{=M expression:expression}
\end{array} \right\} \therefore \, .
$$

Description. The shaded options are available only in the active Debugger. The ellipsis indicates that allowable options may be repeated any number of times (up to a full input line). Different options may also be combined in an input line.

When the Debugger receives a MEMORY command with a string of options, it responds to the options one at a time, from left to right. If there are syntax, semantic, or processing errors, only the leftmost error is reported, and none of the subsequent commands are processed. For this reason, it is generally best to keep your string of options relatively short.

Although the MEMORY command offers a number of options, they break down into three rather simple categories as shown below;

$$
\text{size control} \left\{ \begin{array}{l} \text{!B} \\ \text{!W} \end{array} \right.
$$

$$
\text{address control} \left\{ \begin{array}{l} \text{expression} \\ \text{expression:expression} \\ @ \\ I \\ \text{no entry} \end{array} \right.
$$

$$
\text{alter memory} \left\{ \begin{array}{l} \text{=expression} \\ \text{=M expression} \\ \text{=M expression:expression} \end{array} \right.
$$

The size control options allow you to specify whether a single byte or a two-byte word is to be displayed and/or updated. The !B option requests a byte display; !W requests a two-byte word display. The Debugger initially defaults to a byte display if neither option is specified, and thereafter defaults to the last option specified. Displays always have a hexadecimal format. Thus, the ASCII character A is displayed as 41; the ASCII character 4 is displayed as 34.

Word mode is particularly useful in helping you to read address-type variables or instructions in memory. Since the 8080 or 8085 processor considers the low-order byte of an address variable to be the byte with the lower memory address, a byte display shows the two bytes reversed from the order in which they should be read. Word

mode displays pairs of addresses as sixteen-bit hexadecimal numbers. (Bytes are paired off starting with the specified memory location, whether the address is even or odd.)

The address control options allow you to specify the address portion of the M command. This address, which determines the location to be displayed, may range from 0 through 0FFFF (hexadecimal). You must be certain the current address is valid within your system. Note, for example, that few application systems contain 64K of memory; you must not attempt to display non-existent memory.

The address control options both alter the current address and display the resulting address and its contents. The options have the following meanings:

- Expression—An expression is usually given as a hexadecimal address. Thus, the command M 4000 sets the current address to 4000H and displays the value at that address. The expression may also include a value to be added to or subtracted from the base address. Assume, for example, that from looking at a Task Descriptor, you know that a stack is based at address 40A6 and has a size of 64 bytes (40H). The command M!B 40A6+3F displays the contents of the last byte of the stack.

  Users of the active Debugger have the option of assigning addresses to PL/M-like symbolic addresses. Thus, if the address 4000 is assigned to the symbolic address .A, the command M.A displays the contents of address 4000.

- Expression:expression—This option allows you to specify a range of addresses to be displayed. Thus, the command M 4001:4005 displays the five bytes beginning at address 4001 and continuing through address 4005. By contrast, the command M!W 4001:4005 displays six bytes located at 4001 through 4006 as three four-digit hexadecimal values. Notice that you must be certain that the ending address is greater than the starting address; otherwise, only one item will be displayed.

- @—the @ option makes the contents of the currently displayed location the current address. Thus, if you happen to be looking at the MESSAGE field of a Task Descriptor, the M @ command sets the current address to the address of the message and displays the LINK field of that message. When using the @ option, you must be certain that the new current address is a defined location.

- \—The \ option decrements the current address by one for byte mode (!B specified) or by two for word mode (!W specified). Thus, if the current address is 4005, the command M \\\\\ displays the contents of locations 4004, 4003, 4002, 4001, and 4000 in byte mode, or the contents of locations 4003, 4001, 3FFF, 3FFD, and 3FFB in word mode.

- /—The / option increments the current address by one for byte mode (!B specified) or by two for word mode (!W specified). Thus, if the current address is 4000, the command M ///// displays the contents of locations 4001, 4002, 4003, 4004, and 4005 in byte mode, or the contents of locations 4002, 4004, 4006, 4008, and 400A in word mode.

- no entry—Coding an M command with no options is the same as coding the M command with a single / option. Thus, the commands M and M / are equivalent. If the current address is 4000, both commands display the contents of location 4001 in byte mode, or the contents of location 4002 in word mode.

The alter-memory options are available only with the active Debugger. These options alter and then display the contents of the specified memory locations to verify the change. The alter-memory options rely on previous options for certain

parameters; previous M command options determine whether the operation involves words or bytes and whether a single address or a range of locations is to be modified. The options have the following meanings:

- =expression— This option sets the contents of the current byte or word equal to the value of the expression. If the previous option specified a range of addresses, this option alters the entire sequence of locations. For example, assume that a previous M command specified the range of locations 4000:4005. The command M=30 fills locations 4001 through 4005 with ASCII zeros.

- =M expression—With this option, "expression" must evaluate to an address in memory. The value at that address is then used to alter the current byte or word. If the previous option specified a range of addresses, this option alters the entire sequence of locations.

   For example, assume that a previous M command specified the range of locations 4000:4005 and that location 3000 contains the ASCII character A (41H). The command M=M 3000 moves the letter A (41H) into locations 4001 through 4005.

- =M expression:expression—This option moves the sequence of characters delimited by the two address expressions to the sequence of locations beginning with the current address. Assume, for example, that locations 3000 through 3005 contain the letters ABCDEF, and the current address is 4000. The command M=M 3000:3005 moves the characters ABCDEF to locations 4000 through 4005.

   If the previous M option specifies a range of addresses, that previous option controls the size of the destination field; the source data is truncated or repeated as required. In the above example, if the previous M command specified the addresses 4000 through 4002, the command M=M 3000:3005 moves only the letters ABC. If the previous command specified addresses 4000 through 4009, the command M=M 3000:3005 moves the letters ABCDEFABCD.

NOTE

It is possible to request quite a lengthy display with the M command (M 3000:4000, for example). You can terminate such a display by entering a control-C. This terminates the display at the end of the current output line.

VIEW Command.

Function. The VIEW command displays the contents of a particular RMX/80 system list. For example, the command V R displays the current contents of the Ready List.

Format.

$$
V \left\{ \begin{array}{l} T \\ E \\ R \\ S \\ D \\ M \text{ [expression]} \\ W \text{ [expression]} \\ \text{no entry} \end{array} \right\}
$$

Description. With the passive Debugger, an option must always be specified. If the M or W option is selected, it must be followed by an expression which specifies the address of an Exchange Descriptor.

The VIEW command options have the following meanings:

T = Task List; a list of all tasks in the system.
E = Exchange List; a list of all exchanges in the system.
R = Ready List; a list of all tasks currently on the Ready List.
S = Suspend List; a list of all tasks currently suspended.
D = Delay List; a list of all tasks currently in a timed wait.
M = Message List; a list of all messages queued at a particular exchange.
W = Wait List; a list of all tasks queued at a particular exchange.
no entry = Ready, Suspend, Delay, Wait, and Message lists.

The VIEW command generates displays of lists of addresses. These lists provide useful information about the state of the system. For example, the V S command lists the Task Descriptor addresses of all currently suspended tasks.

The general format of the VIEW command display is as follows:

cL=xxxx, . . .,xxxx

The lower-case c is replaced by the list option specified in the VIEW command. Thus, the display for a V R command begins with the characters RL= to indicate that the display is for the Ready List. Each xxxx is replaced by the address of the Task Descriptor of a ready task.

The VIEW command produces one other important piece of data: when displaying a list of Task Descriptor addresses, the VIEW command places an asterisk after the address of a Task Descriptor if that task has had a stack overflow or stack full condition since the last time the system was restarted. This feature is an aid in determining stack size requirements, as discussed in Chapter 3.

With the passive Debugger, the V M and V W commands require the use of an expression that yields the address of an exchange. The Debugging issues an IN-VALID ADDRESS message if the expression does not yield a valid Exchange Descriptor address. The use of the expression is optional with the active Debugger (see below).

When the VIEW command is executed, a high-priority Debugger task takes a "snapshot" of the requested list(s) and then gives control to a lower-priority task whose responsibility is to display the list(s). The high-priority task will cause a slight increase in interrupt latency. The displayed list(s) may be obsolete by the time the display is completed, because the system continues to run.

With the active Debugger, you are not required to include the address of an exchange in the V M or V W commands. You may, of course, use this option to view the tasks or messages waiting at a particular exchange. When no expression is given, the Debugger provides a display for each Exchange Descriptor in the system. The display format is somewhat modified to indicate the exchange being displayed:

cL(yyyy)=xxxx, . . .,cL(yyyy)=xxxx, . . .,xxxx; . . .

The lower-case c is replaced by an M or a W, depending on which command you entered; yyyy is replaced by the address of the Exchange Descriptor being displayed; and xxxx is replaced by the address of a message or Task Descriptor queued at the exchange.

With the active Debugger, you can enter a V command without any options. The Debugger displays the Ready List, Suspend List, and Delay List, the Wait lists, and the Message lists.

NOTE

A single VIEW command never displays more than 100 addresses. Thus, a
V W, V M, or (particularly) a V command might not display all the Wait or
Message Lists. The V E command reveals this situation when it occurs, since
V E then lists more exchanges than displayed by the earlier V command.
You can use additional V W or V M commands to display the remaining
Wait or Message Lists by requesting them one at a time.

FORMAT Command.

Function. The FORMAT command formats and displays the contents of a Task
Descriptor, Exchange Descriptor, or message.

Format.

$$F \begin{bmatrix} T\text{ [expression]} \\ E\text{ [expression]} \\ M\text{ expression} \end{bmatrix}$$

Description. The expression in an F T (Format Task Descriptor) command must
yield the address of a Task Descriptor. Similarly, the expression in an F E (Format
Exchange Descriptor) command must yield the address of an Exchange Descriptor.
The Debugger issues an INVALID ADDRESS message if the expression is
improperly stated.

The expression in an F M (Format Message) is not checked for validity. The Debugger
will format whatever data happens to begin at the specified location as a message.
However, you should be certain that the expression yields an address that is valid
within your system. You must not instruct the Debugger to fetch data from memory
that is not used in your system.

With the active Debugger, it is not necessary to enter an expression with an F T or
F E command. In this case, the Debugger displays the first Task Descriptor or Ex-
change Descriptor, depending on the command, on its list of Task or Exchange
Descriptors. When no options are specified, the next descriptor in the current list is
displayed. If all the descriptors on a list have been displayed, the Debugger issues an
error message and then issues a prompt for another command.

Formatted Task Descriptor Display. The Debugger formats the fields of a Task
Descriptor in the same order as described under "Details of System Operation" in
Chapter 2. The Debugger adds three fields: the first display field specifies the ad-
dress of the Task Descriptor; the last two display fields contain the task name and
the number of bytes remaining on the task's stack, respectively.

The format is as follows:
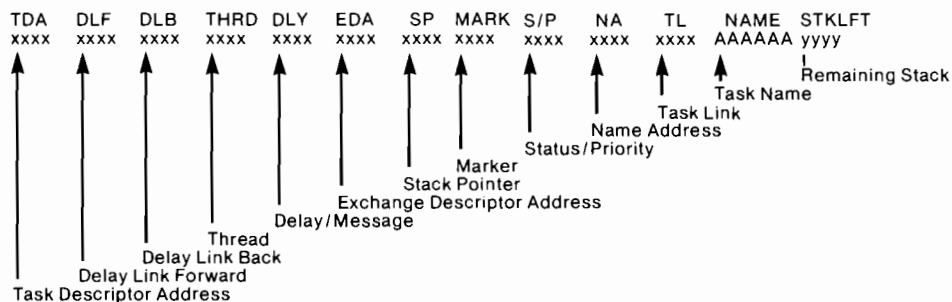
The STKLFT field indicates the number of never-used bytes remaining in the stack when this FORMAT command is executed.

You can determine whether a task is suspended (or preempted or delayed) by looking at the status bits (leftmost two digits of the S/P field). You can determine if the task is waiting by noting the EDA field, then using the VIEW command to check the waiting task list at that exchange.

Formatted Exchange Descriptor Display. The Debugger formats the fields of an Exchange Descriptor in the same order as described under "Details of System Operation" in Chapter 2 of this manual, with one exception: the first display field specifies the address of the Exchange Descriptor.

The format is as follows:

```
EDA      MH       MT       TH       TT       EL
xxxx     xxxx     xxxx     xxxx     xxxx     xxxx
 ↑        ↑        ↑        ↑        ↑         Exchange Link
 |        |        |        |        Task Tail
 |        |        |        Task Head
 |        |        Message Tail
 |        Message Head
 Exchange Descriptor Address
```
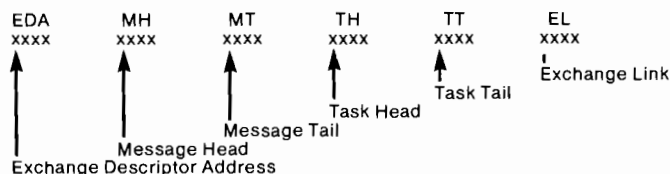
Formatted Message Display. The Debugger formats the fields of a message in the same order as described under "Message Coding" in Chapter 2 of this manual, with one exception: the first display field specifies the address of the message.

The format is as follows:

```
MDA  LINK  LEN   TYPE  HEA   REA   REM
xxxx xxxx  xxxx  xx    xxxx  xxxx  xx    xx  xx  xx  xx  xx  xx  xx
 ↑    ↑     ↑     ↑     ↑     ↑     ↑     xx  xx  xx  xx  xx  xx  xx
 |    |     |     |     |     |     Remainder
 |    |     |     |     |     Response Exchange Address
 |    |     |     |     Home Exchange Address
 |    |     |     Type
 |    |     Length
 |    Link
 Message Address
```
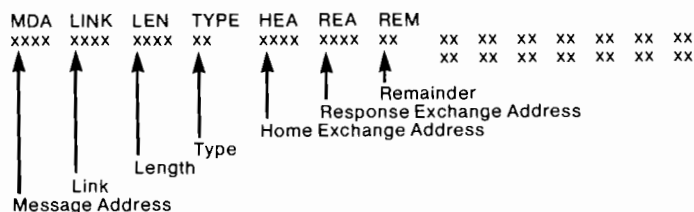
A message display is limited to 20 lines of text or 160 bytes of the Remainder field. You can terminate this display by entering a control-C. Control-C terminates the display at the end of the current line.

QUIT Command.

Function. The QUIT command terminates the debugging session. Notice that all terminal input is directed to the Debugger during a debugging session. The QUIT command directs all subsequent terminal input to your system rather than to the Debugger.

Format.

Q

Description. The QUIT command has no options.

Although the QUIT command terminates this debugging session, the Debugger remains available for use. You need only enter a control-C to re-activate the Debugger.

NOTE

The Terminal Handler provides controls to prevent user output to the terminal during a debugging session. These controls should usually be reversed when the debugging session is terminated. If control-S has been used to delay user output, control-Q resumes the output. If control-O has been used to discard user terminal output, another control-O resumes the output.

NOTE

All the remaining Debugger commands apply only to the active Debugger. Attempting to use any of the following commands with the passive Debugger causes a syntax error message.

EXECUTE Command.

Function. The EXECUTE command allows you to execute, on command, a procedure in memory including any of the RMX/80 operations or a user-supplied procedure.

Format.

X procedure-address [(parameter-1, . . ., parameter-n)]

Description. The procedure-address in an EXECUTE command may be the address of an RMX/80 operation or of one of your own system's procedures. Parameter requirements, if any, depend on the procedure being called. As examples, the RQSEND operation requires two parameters (an exchange address and the address of the message to be sent); but the RQDTSK operation requires only one (the address of the Task Descriptor to be deleted). All parameter passing conforms to the rules for calling a PL/M procedure from an assembly language program module. (Refer to "Parameter Passing and Returned Values" in Chapter 3.) "Procedure-address" may be the address of an assembly language subroutine, provided that the PL/M parameter passing conventions (as defined under "Parameter Passing and Returned Values" in Chapter 3) are followed.

The EXECUTE command sets up the parameters provided, and then calls the procedure. Upon return from the called procedure, the task registers contain any values returned by the called procedure, and the Breakpoint Task becomes undefined (BT =0). The Breakpoint Task is explained under the CHANGE command. The values returned can be displayed by using the task-register version of the DISPLAY command.

The EXECUTE command can be used to execute RQSUSP, in order to suspend tasks irrelevant to the part of the system you are trying to debug. This has the advantage of simplifying the system. However, you must be certain that the RQSUSP operation is available. RQSUSP is not normally linked into the system as part of the RMX/80 Nucleus; the operation is available only if you use it in one of your tasks or explicitly link it into your system.

For the iSBC 80/10 version, note the restriction that if the Debugger's priority is in the range 0 to 128, the EXECUTE command should not be used.

DEFINE Command.

Function. The DEFINE command allows you to assign PL/M-like symbols (numeric variables) to frequently-used memory addresses or other numeric values.

Format.

    D numeric-variable = expression

Description. The ability to assign symbolic names to frequently-used addresses is particularly useful in debugging, since many of these addresses change each time the system is relinked and relocated. The DEFINE command allows you to refer to these addresses using PL/M-like symbols, which are easier to remember than addresses and which need not change as the program changes. All you need do is enter the appropriate DEFINE commands at the beginning of each debugging session.

"Numeric-variable" is an identifier. This identifier must conform to rules that fall part-way between PL/M and assembly language. The identifier must begin with a period and must not exceed six characters, not counting the initial period and any dollar signs. The first character must be alphabetic, and the remaining characters may be either alphabetic or numeric.

The expression in the format should yield an address or other value meaningful to you in debugging. The DEFINE command allows you to retain the same symbolic name from session to session, providing the system continues to run and is not restarted.

The DEFINE command does not allow duplicate numeric variable symbols. The Debugger issues a DUPLICATE SYMBOL message if you try to define the symbol more than once. The CHANGE command allows you to assign a new address to a numeric variable defined in this debugging session.

REMOVE Command.

Function. The REMOVE command allows you to remove a previously-defined numeric variable from the system. Thus, the REMOVE command is associated with the DEFINE command, previously described.

Format.

    Z [numeric-variable]

Description. As in the DEFINE command, "numeric-variable" must be a PL/M-like symbol. If no such symbol has been defined, the Debugger issues an error message followed by a prompt for a new command.

A Z (REMOVE) command without any parameter removes all previously defined numeric variables from the system. Obviously, care must be taken when issuing Z without a parameter.

Debugger Breakpoint Feature. The final four commands—CHANGE, DISPLAY, GO, and SCAN—are concerned with the Debugger breakpoint feature. To clarify the descriptions of these commands, a general explanation of Debugger breakpointing is provided here.

The Debugger provides eight *breakpoint registers,* which are identified by the symbols B0, B1, . . ., B7. Each breakpoint register may be used to store the address of an instruction or exchange that is to cause a breakpoint. The breakpoint-register format of the CHANGE command allows you to set, change, or clear a breakpoint.

The Debugger recognizes three types of breakpoints: *execution breakpoints, send breakpoints,* and *wait breakpoints.* An execution breakpoint occurs when a task executes an instruction whose address is specified in a breakpoint register. A send breakpoint occurs when a task sends a message to a specified exchange. A wait breakpoint occurs when a task waits at a specified exchange and subsequently receives a message. Send and wait breakpoints are also referred to as *exchange breakpoints.*

When a breakpoint occurs, the Debugger temporarily removes from the system the task that caused the breakpoint and issues a display message to the terminal in order to identify the breakpoint (by register) and the task. If the breakpoint register was set specifying the Continue option, the task is automatically returned to the system after the output message is issued. If the Continue option was not specified, the task remains outside the system until you restart it from the terminal with the GO command.

Each time an execution breakpoint occurs, the Debugger cancels the breakpoint by resetting the breakpoint register to zero and restoring the breakpointed instruction to its original state. Unlike execution breakpoints, however, exchange (send or wait) breakpoints are not reset each time a break occurs.

The Debugger suspends tasks not by using the RQSUSP operation, but by placing their Static Task Descriptor addresses on a list of its own called the *Breakpoint List.* This list is similar to the RMX/80 Suspend List in that it removes the task from contention for system resources. The Breakpoint List includes all tasks that have incurred breakpoints (without the Continue option) and have not yet been resumed, and also any tasks that have incurred a stack overflow while an overflow scan was in effect (see description of the SCAN command). Each task on the Breakpoint List has a state—Execute, Wait, Send, or Overflow. The first three states correspond to the type of breakpoint, and the last designates a stack overflow detected by the SCAN command.

You may designate one task on the Breakpoint List as the *Breakpoint Task.* Doing this (via the breakpoint-task format of the CHANGE command) enables you to examine and alter the contents of that task's registers, which are stored on the task's stack. You use the task-register format of the DISPLAY command to display the contents of one or more task registers, and the task-register format of the CHANGE command to alter the contents of a task register.

You may resume a task on the Breakpoint List by making it the Breakpoint Task and then using the GO command. (This is true of tasks that have incurred a stack overflow detected by SCAN as well as those that have incurred breakpoints.)

NOTE

There are two differences between the RMX/80 Debugger execution breakpoints and ICE-80 or ICE-85 breakpoints that may cause confusion. Both of these differences are based on the fact that the RMX/80 Debugger replaces the first byte of the instruction to be breakpointed with an RST 5 instruction. The following conditions should be noted:

- Setting a breakpoint at the location of the breakpoint just incurred and then continuing (with the Go Command) will *always* result in the task incurring another breakpoint.

- If RMX/80 "crashes" while an execution breakpoint is set, the RST 5 will be left in the code. The code must, therefore, be "patched" with the original instruction, or the program must be reloaded.

CHANGE Command.

Function. The CHANGE command allows you to establish or change a breakpoint, define a Breakpoint Task, alter the Breakpoint Task's registers, or change the value of a numeric variable previously entered via a DEFINE command.

General Formats.

$$
\text{breakpoint-register} \quad = \quad \left\{ \begin{array}{l} \text{expression} \quad \left[\begin{array}{l} E[C] \\ \left[W[C] \right] \cdots \\ S[C] \end{array} \right] \\ \text{zero-expression} \end{array} \right\}
$$

$$
\left\{ \begin{array}{l} BT \\ \text{task-register} \\ \text{numeric-variable} \end{array} \right\} \quad = \quad \text{expression}
$$

Description of Breakpoint-Register Format.

$$
\text{breakpoint-register} \quad = \quad \left\{ \begin{array}{l} \text{expression} \quad \left[\begin{array}{l} E[C] \\ \left[W[C] \right] \cdots \\ S[C] \end{array} \right] \\ \text{zero-expression} \end{array} \right\}
$$

The ellipsis in the format specifies that the indicated portion of the format may be repeated. In practice, this means that both S and W options may be specified in one command. The C option, if present, must not be separated by a space from the E, W, or S which precedes it.

This format sets or clears a breakpoint, using one of the eight breakpoint registers. "Breakpoint-register" must be the symbol for a breakpoint register (B0, B1, . . ., or B7). To set a breakpoint, you specify, after the equals sign, an expression that yields the address of an instruction or an Exchange Descriptor—followed by a code for the type of breakpoint—E (execution), S (send), W (wait). To cancel a previously set breakpoint, you specify a zero-expression (an expression that yields zero).

The breakpoint codes have the following meanings:

E: Execution breakpoint—break when the specified instruction is executed.

W: Wait breakpoint (an exchange breakpoint)—break when a task is removed from the specified exchange.

S: Send breakpoint (an exchange breakpoint)—break when a task sends a message to the specified exchange.

As mentioned previously, the Debugger temporarily removes from the system a task that causes a breakpoint. The Continue (C) option, which applies to the particular E, W, or S option immediately preceding it, allows the breakpointed task to resume contention for system resources automatically, as soon as a display message has been sent to the terminal. When this option is not specified, the task remains outside the system until you restart it by using the "BT=" format of the CHANGE command and a GO command.

For an execution breakpoint, "expression" must yield the address of an instruction in RAM. The Debugger places a RESTART instruction at this address. (The Debugger also saves the original contents of this address so that the instruction can be restored when the breakpoint is cancelled.) The expression must yield the first byte of the instruction; if the RESTART instruction is embedded within a multi-byte instruction, the instruction will malfunction when executed. Each time an execution breakpoint occurs, the Debugger cancels the breakpoint by resetting the breakpoint register to zero and restoring the breakpointed instruction to its original state.

An execution breakpoint generates the following display on the terminal:

    Bn,E[C],TDA=xxxx

If the Debugger is not active when a task incurs a breakpoint, the Debugger is automatically activated (as if control-C were struck). The n specifies the number of the breakpoint register; E identifies an execution breakpoint; the C appears only if the Continue option was specified for this breakpoint; xxxx is replaced by the address of the Task Descriptor of the task that caused the breakpoint.

For the iSBC 80/10 version, note the following restriction: if a task which has a software priority in the range 0 to 128 is breakpointed with an execution breakpoint, it should not be resumed. Resuming such a task may cause incorrect execution.

For an exchange (send or wait) breakpoint, a W, S, or both may follow the expression. The expression must yield the address of an Exchange Descriptor, or an error message is generated. If W is specified, any task that receives a message or times out after performing an RQWAIT at the specified exchange is breakpointed. If S is specified, any task that performs an RQSEND to the exchange is breakpointed. Unlike execution breakpoints, exchange breakpoints are not reset each time a break occurs. Note that both the W and S breakpoint types may be specified for the same exchange breakpoint. If neither breakpoint type is specified, W is assumed.

Several additional restrictions apply to exchange breakpoints.

•   A task that performs an RQACPT at a breakpointed exchange always receives a zero return value.

•   An RQSUSP operation for a task waiting at a breakpointed exchange is ignored.

•   An RQRESM operation for a task waiting at a breakpointed exchange causes the breakpoint facility to malfunction.

•   An S type breakpoint should not be specified for an interrupt exchange.

- Use the VIEW command to examine the message list or task list of a breakpointed exchange. A FORMAT or MEMORY command used for this purpose will display breakpoint control structures rather than the actual message or task list.

- An exchange breakpoint does not detect a task that performs an RQWAIT with a time limit of one time unit if the wait times out before a message is sent to the exchange.

- Exchange breakpoints increase both the interrupt latency and the total load on the system. In extreme cases, this may cause a missed interrupt.

An exchange breakpoint generates the following display on the terminal:

$$\text{Bn,} \quad \left\{ \begin{array}{c} \text{W} \\ \text{S} \end{array} \right\} \quad \text{[C],TDA=xxxx,MSGA=aaaa}$$

If the Debugger is not active when a task incurs a breakpoint, the Debugger is automatically activated, and the display above is given followed by the Debugger prompt. The n specifies the number of the breakpoint register; W or S appears depending on whether the breakpoint was for a wait or a send; the C appears only if the Continue option was specified for this breakpoint; xxxx is replaced by the address of the Task Descriptor of the task that caused the breakpoint; aaaa is replaced by the address of the message sent to or received from the breakpoint exchange.

Description of Breakpoint-Task Format.

BT=expression

This format of the CHANGE command allows you to select a particular task from the Breakpoint List as the current Breakpoint Task. (Note that there may be only one Breakpoint Task at a time.) The expression must yield the address of a Task Descriptor on the Breakpoint List. (The DISPLAY command can display the current contents of the list.) An error message is generated if the expression does not yield the address of a Task Descriptor on the Breakpoint List.

When the Debugger removes a Breakpoint Task from the system, it performs a context save, thus preserving the contents of the task's registers. The DISPLAY command gives you access to the current Breakpoint Task's registers. The "task-register = expression" format of the CHANGE command allows you to alter those registers. For execution breakpoint tasks, all the the registers may be meaningful and are displayed as they were when the breakpoint occurred. For exchange breakpoints, only the Stack Pointer and Program Counter registers are meaningful, since the other registers are modified in the RQWAIT and/or RQSEND operations. These other registers all contain zeros. Also, the Program Counter always contains the address of the next instruction after the RQWAIT or RQSEND operation. For breakpoints caused by a stack overflow (see the SCAN command), none of the registers are meaningful, so they are all set to zeros.

A task must be designated as the current Breakpoint Task before the GO command (described later in this chapter) can be used to return it to the system.

Description of Task-Register Format.

task-register=expression

This format of the CHANGE command updates the contents of the specified Breakpoint Task register with the value of the expression. For exchange breakpoints, altering any register other than the Stack Pointer or Program Counter has no effect. For overflow breakpoints, altering a register has no effect.

Task-register must be one of the following values:

RA	A register (accumulator)
RB	B register
RC	C register
RD	D register
RE	E register
RH	H register
RL	L register
RF	Flags
RPCH	High-order byte of Program Counter
RPCL	Low-order byte of Program Counter
RSPH	High-order byte of Stack Pointer
RSPL	Low-order byte of Stack Pointer
RBC	B and C register pair
RDE	D and E register pair
RHL	H and L register pair
RPC	Program Counter
RPSW	Program Status Word (accumulator and flags)

Description of Numeric-Variable Format.

    numeric-variable=expression

The DEFINE command allows you to assign symbolic names to addresses required during debugging sessions. This format of the CHANGE command allows you to alter the value assigned to a previously-defined numeric variable. An error message is generated if ''numeric-variable'' is not previously defined. The format of ''numeric-variable'' is as described under the DEFINE command.

The Debugger allows up to 30 numeric variable definitions.

DISPLAY Command.

Function. The DISPLAY command generates displays of various Debugger structures associated with Breakpoint Tasks, and the values of symbols assigned via the DEFINE command. You do not enter an op-code to invoke the DISPLAY command; instead, you simply enter the name of the item to be displayed.

Format.

$$
\left\{
\begin{array}{l}
\text{breakpoint-register} \\
\text{BT} \\
\text{BL} \\
\text{B} \\
\text{R} \\
\text{task-register} \\
\text{numeric-variable} \\
\text{no entry}
\end{array}
\right\}
$$

Description. To specify one of the breakpoint registers, enter the name of the desired register, B0, B1, . . ., B7. The DISPLAY command generates the following display on the terminal:

    Bn=xxxx op op

The n in the display is replaced by the number of the specified breakpoint register; xxxx is replaced by the address currently associated with this breakpoint register (0 if the breakpoint is currently reset); op is replaced by the symbols for a breakpoint option, as explained under the CHANGE command.

Enter BT to display the address of the current Breakpoint Task's Task Descriptor. The DISPLAY command generates the following display:

    BT=xxxx(s)

The xxxx in the display is replaced by the address of the Task Descriptor of the current Breakpoint Task (0 if there is no current Breakpoint Task). The s is replaced by one of the following letters to indicate the task's status: O = Stack Overflow; E = Executing; W = Waiting; and S = Sending.

Enter BL to display the entire Breakpoint List. The Breakpoint List contains the Task Descriptor address and current status of each breakpointed task. The display has the following format:

    BL=xxxx(s),xxxx(s), . . .,xxxx(s)

Each xxxx(s) is replaced with the Task Descriptor address and status (Stack Overflow, Executing, Waiting, or Sending) of a breakpointed task.

Enter B to obtain a display of all the Breakpoint Registers followed by the Breakpoint Task address and a display of the Breakpoint List. The display has the following format:

    B0=xxxx op op
    B1=xxxx op op
            .
            .
            .
    B7=xxxx op op
    BT=xxxx(s)
    BL=xxxx(s),xxxx(s), . . .,xxxx(s)

Enter R to obtain a display of the current Breakpoint Task's registers. The display has the following format:

In the display, xx and xxxx are replaced by current contents (in hexadecimal format) of the register or register pair. If desired, you can display the contents of any individual register by entering a task-register name from the following list:

RA	A register (accumulator)
RB	B register
RC	C register
RD	D register
RE	E register
RH	H register
RL	L register
RF	Flags
RPCH	High-order byte of Program Counter
RPCL	Low-order byte of Program Counter
RSPH	High-order byte of Stack Pointer
RSPL	Low-order byte of Stack Pointer
RBC	B and C register pair
RDE	D and E register pair
RHL	H and L register pair
RPC	Program Counter
RPSW	Program Status Word (accumulator and flags)

Enter the name of a numeric variable (see DEFINE command) to obtain its current value. The display has the following format:

.abcdef=xxxx

In the display, .abcdef is replaced by the name of the numeric variable; xxxx is replaced by the value currently assigned to the numeric variable.

To obtain a display of the current values of all currently defined numeric variables, strike the Carriage Return without making any other entry. Because numeric variables are commonly used to assign symbolic labels to frequently used addresses in a debugging session, this option provides a handy reminder. If there are no currently defined numeric variables, the display consists only of another prompt.

GO Command.

Function. The GO command resumes the Breakpoint Task—i.e., makes it ready to run again.

Format.

G

Description. Options are not permitted with the GO command.

The GO command resumes the Breakpoint Task by removing it from the Breakpoint List and returning it to the system, so that it becomes ready to run again. When it becomes the highest priority ready task, it will begin running at the next instruction that was to be executed when the break occurred.

As a side effect, the GO command causes the Breakpoint Task to become undefined (BT=0).

If the GO command is executed when there is no Breakpoint Task, the Debugger issues an error message and issues a prompt for another command.

See the description of the CHANGE command for a description of how to set a breakpoint.

SCAN Command.

Function. The SCAN command periodically checks the Task Descriptors of all tasks in the system to determine whether any task's stack has an overflow condition. If one is found, the SCAN command issues an overflow message to identify the task with the overflow condition.

Format.

<div align="center">

S [expression]

</div>

Description. The expression in the SCAN command serves two purposes. When provided, the expression specifies the frequency of the scan. Termination of the scan is accomplished by omitting the expression.

The expression specifies the number of system time units between scans. (A system time unit is 50 msec in the iSBC 80/20 or 80/30 version. Thus, an expression with the value 20 specifies a scan once each second. On the iSBC 80/10, the length of a system time unit may vary; refer to Appendix G.) If the value of the expression is too great, a stack overflow may cause a fatal system malfunction before the Debugger detects the overflow. If the value of the expression is too small, the scan may occur so frequently that it interferes with the operation of the system. Fortunately, the interactive nature of the Debugger allows you to change the scan rate at any time. Therefore, it is better to start with a frequent scan rate so that you are more likely to catch any stack overflow. If the scan rate seems to interfere with the system's performance, you can alter the rate.

When the SCAN command detects a stack overflow, it generates the following display on the terminal:

OVERFLOW,TDA=xxxx

TDA stands for Task Descriptor Address; xxxx is replaced with the address of the Task Descriptor for the task with the stack overflow.

The SCAN command removes from the system a task with stack overflow by adding it to the Debugger's Breakpoint List. You can resume execution of the task with the GO command.

The SCAN command cannot suspend a task that is waiting at an exchange, since this might destroy synchronization in some systems (for example, when two tasks alternately wait at the same exchange). Instead, the SCAN command issues the message MORE OVERFLOW(S). As soon as the task's wait is satisfied, the SCAN command moves the task to the Breakpoint List and issues an overflow message. Use the VIEW command to determine which tasks have the overflow condition. Because of the treatment of waiting tasks, the SCAN command may have residual effects even after you terminate the scanning process. If the scan detects a waiting task with an overflow but you terminate the SCAN command before a message can be issued, the task is moved to the Suspend List when its wait is satisfied. You can detect this condition with the VIEW command.

You can leave the SCAN command in effect even when you terminate a debugging session (see the QUIT command). If an overflow is detected, the SCAN command activates the Debugger. This allows the SCAN command to display the appropriate message(s) and also allows you to enter a response through the terminal.

Error Messages

After you enter a command line, the Debugger checks the command for any errors. When an error is detected, the Debugger aborts the processing of the command, issues an error message, and then issues a prompt for a new command.

The Debugger recognizes three general types of errors:

*   Syntax errors caused by the incorrect specification of a command.
*   Semantic errors caused by such things as an expression yielding an invalid address. For example, certain expressions must yield the address of a Task Descriptor.
*   Processing errors detected during the execution of a command. For example, symbol table space may be exhausted.

An error message consists of two lines:

1. The erroneous line up to the point of the error. A crosshatch (#) follows the character where the Debugger detected the error.

2. A message describing the error. The following is a complete list of messages generated by the Debugger:

        SYNTAX ERROR
        INVALID ADDRESS
        DUPLICATE SYMBOL
        NOT FOUND
        TABLE FULL
        NO BREAKPOINT TASK

Suggestions for Debugging Your System

Debugger Priority. You must choose the priority at which the Debugger runs. Commonly, the Debugger is assigned a priority lower than user tasks. In complex systems, this may not provide adequate Debugger response times. Some experimentation may be required to find a priority that provides adequate response time without seriously affecting the performance of the application software.

A Systematic Approach to Debugging a New System. As mentioned previously, the Debugger runs under the RMX/80 Nucleus and requires the services of the Terminal Handler. By implication, the Debugger requires at least a minimally operational system. If your tasks force the system into an unrecoverable error, the Debugger also becomes inoperative. (ICE-80 or ICE-85 may be used in this situation.)

Careful planning can prevent untested tasks from causing serious errors. When you are certain that your hardware configuration works properly, you can link together your application code with the RMX/80 Nucleus, Debugger, Terminal Handler, and any other RMX/80 tasks needed in your system. However, it is recommended that you name only RMX/80 tasks in your configuration module. (Your Initial Exchange Table may name all the exchanges used in the system, but only the required RMX/80 tasks should be named in the Initial Task Table.)

If you have already implemented the Demonstration System, then the RMX/80 Nucleus, Terminal Handler, and Debugger have already been tested in your hardware environment and are fully operational. The Debugger allows you to add your tasks to the system one by one. You must do this carefully.

First, you add your application task with the highest priority. Keep in mind that RMX/80 always gives control to the highest-priority task when the system is initialized (each time the system is restarted). When none of your application tasks is

included in the Initial Task Table, the highest-priority task is always a pre-tested RMX/80 task. Therefore, the system must be operational. (Actually, nothing is running, since all RMX/80 tasks are waiting to service one of your tasks or an interrupt. The system simply enters a HALT state.) The active Debugger's EXECUTE (X) command allows you to enter an RMX/80 RQCTSK operation. Use this operation to activate your highest-priority application task.

Using the Debugger EXECUTE command, enter the addresses of the RQCTSK operation and the Static Task Descriptor for your highest-priority task. At this point, all the RMX/80 tasks are waiting to service your application tasks, but only one of your tasks exists in the system. Your highest priority task should "run to completion"—in other words, until it waits at an exchange, suspends itself, or deletes itself.

Now you must decide which task to activate next. Typically, this will be the second highest-priority user task, since RMX/80 schedules tasks for execution by priority. In effect, you are manually simulating RMX/80's startup procedure. This puts you in the position of making decisions normally handled by RMX/80. This also gives you a tremendous advantage when debugging: if the system fails as you add your tasks to the system, you know which task directly or indirectly caused the problem.

Using the Debugger to Improve System Performance. Using the Debugger in an operational system can be especially useful for "fine-tuning" the system. For example, you may find that a particular exchange frequently has a relatively large number of messages queued. Raising the priority of the task(s) that service the exchange may improve overall system performance. Similarly, when a message-consuming task spends too much time waiting for a message, system performance may be improved by raising the priority of the message-producing task(s).

Tips on Real-Time Debugging. Certain practices will be helpful to you as you debug your system:

- Become thoroughly familiar with all the Debugger commands.

- Use figure 2-8 as a reference during debugging.

- Keep a list of commonly used addresses, including the addresses of the RMX/80 operations and of your system's Task Descriptors and Exchange Descriptors. Those addresses that are public (the addresses of RMX/80 operations and PUBLIC exchanges) are supplied in the output from the ISIS-II LOCATE program. Addresses that are not public can be obtained by using the Format Task Descriptor (FT) option of the FORMAT command.

- Use the DEFINE command to assign symbols to commonly used addresses.

The most common cause of system failure is writing over a system structure. This kind of failure can be quite subtle. To cite an unlikely example, your program might over-write a Task Descriptor's Stack Pointer address of 3230H with the data 3130H. Such an error might allow the system to run indefinitely without a failure. In general, the Debugger cannot detect such a logic error in an application program; however, the Debugger can detect a stack overflow, a likely side effect of the above problem.

Always keep in mind that your application system continues to run while you use the Debugger. Thus, the Debugger always displays memory as it was when the display was requested; therefore, Debugger displays are always somewhat behind what is actually happening in the system. This is especially important to remember if you modify memory. For example, you might format and display a message. If you decide to alter some portion of the message, it is quite possible that the system will write a different message into the same memory before you can enter your modification. The Debugger breakpoint facility and the X (Execute) command can be used to restrict the dynamic nature of the system.

On the subject of modifying memory, there are, of course, certain fields that you should never modify. Most of these are simply a matter of common sense. For example, you should not modify the various linkage fields supplied by RMX/80. (The Delay Link Forward, Delay Link Back, and Status fields of the Task Descriptor and the Link field of a message are prime examples.) A less obvious example is the Priority field of the Task Descriptor. Keep in mind that tasks are queued onto the Ready List according to priority. If you alter the priority of a task already on the Ready List, you may destroy the order of the list. (Task priority is considered only when a task is being entered on the list. RMX/80 cannot tell if you change the priority of a task already on the list.) An out-of-sequence Ready List can seriously affect system performance.

You must also remember that a memory change in a debugging session does not affect your system except during that session. If you want to make the change permanent, you must alter your source program(s) and recompile or reassemble, link, locate, and test the resulting object file.

Examples of Debugger Command Usage

Setting an Execution Breakpoint. Assume that a task is executing code in RAM between the addresses E242H and E2F0H. (The RAM may be either in the iSBC system or in a portion of the Intellec Development System used to emulate iSBC memory.) The following commands set breakpoints for the instructions at addresses E250 and E263:

    B0=E250 E
    B1=E263 E

The Debugger removes the task from the system if it attempts to execute either of these instructions. For example, assume that the task has its Task Descriptor at C100H. If the task attempts to execute the instruction at E263, the Debugger issues the following message:

    B1,E,TDA=C100

The following command makes this task the Breakpoint Task:

    BT=C100

Making a task the Breakpoint Task allows you to interact with the task. For example, you might enter the following DISPLAY command to examine the contents of the task's registers:

    R

At this point, you might decide to zero the task's accumulator and then return it to the system with the following commands:

    RA=0
    G

Setting an Exchange Breakpoint. Assume that there is an Exchange Descriptor at location B212, and you want to breakpoint any task that waits for a message at that exchange. The following command sets the breakpoint:

    B4=B212 W

Now assume that the task described in the previous example performs a wait at this exchange. When the task receives a message from this exchange, the Debugger suspends the task and issues the following message:

B4,W,TDA=C100,MSGA=aaaa

This message indicates that the task waiting at the Exchange Descriptor at B212 received a message and has been removed from the system.

Breakpointing Multiple Tasks. It is frequently useful to debug the interaction between two tasks. Assume, for example, that two tasks communicate via the exchange described in the previous example and you want to breakpoint them when a message is passed between them. You should specify:

B4=B212 S W

Assume further that a task whose Task Descriptor is at A100 waits at the exchange, and a task whose Task Descriptor is at A200 sends a message to the exchange. The Debugger removes both tasks from the system and issues the following messages:

B4,S,TDA=A200,MSGA=aaaa
B4,W,TDA=A100,MSGA=aaaa

The first display indicates that the task whose Task Descriptor is at A200 performed a send to the breakpoint Exchange Descriptor and gives the address of the message. The second display indicates that the other task was removed from the system immediately after it received the same message. At this point, you can select either task to be the Breakpoint Task by naming the task's Task Descriptor in the "BT=" form of the CHANGE command.

Configuration, Linking, and Locating

This section supplies the information you need to include the Debugger in your configuration module, and to link and locate the resulting object code. Instructions for preparing the configuration module from this information are given in Chapter 3, along with general instructions for linking and locating your system.

Configuration Requirements

Terminal Handler. Any system that uses the Debugger must also include the full input-output version of the Terminal Handler. For Terminal Handler configuration requirements, refer to Chapter 4.

Tasks. One task must be defined — RQADBG for the active Debugger, or RQPDBG for the passive Debugger. For either version, the stack length is 64 and the default exchange is RQWAKE.

You may assign the Debugger any priority that is appropriate within your system. Note that the specified priority will be the priority of the task that communicates with the Terminal Handler; the Debugger creates additional tasks that may have different priorities. In particular, the stack overflow scan and the tasks that handle exchange breakpoints both run at priority 0.

Initial Exchanges. The Debugger uses the RQDBUG, RQALRM, and RQWAKE exchanges. However, these are also configuration requirements for the input-output Terminal Handler, which must always be included with the Debugger. No additional exchanges are required by the Debugger.

PUBLIC Data. No PUBLIC data items need be defined at configuration time for the Debugger.

Linking and Locating

The following files must be added to the list of files given to the LINK program to link the Debugger into your system:

1.   ADB820.LIB (for the iSBC 80/20), ADB830.LIB (for the iSBC 80/30), or ADB810.LIB (for the iSBC 80/10)

2.   PDB820.LIB (for the iSBC 80/20), PDB830.LIB (for the iSBC 80/30), or PDB810.LIB (for the iSBC 80/10)

Both files must be given in the order shown for the active Debugger. The first file should be omitted for the passive Debugger, to prevent additional code from being linked in.

Recall that you must also link in the input-output version of the Terminal Handler. Moreover, the LOCATE program may detect apparent memory conflicts, and it then prints error messages, which can be ignored. For a full discussion of these matters, refer to the section of Chapter 3 that deals with linking and locating.

General Description

The Disk File System (DFS) provides disk access capabilities to RMX/80 users. (Throughout this chapter the term "disk" refers to Intel diskette products.) The services are functionally similar to the disk facilities available in ISIS-II, although they operate in the real-time environment supported by RMX/80. Files may be created, deleted and changed; data may be accessed sequentially and directly ("randomly"). Many applications do not need all the services which DFS offers; the modular design of the system allows DFS functions to be implemented selectively, thereby keeping memory requirements consistent with the needs of the application. The Disk File System also gives the user the flexibility to configure various complements of Intel disk drives and controllers, again according to the needs of the application.

DFS Capabilities

Available Services. Table 7-1 summarizes the services offered by DFS.

Table 7-1. Disk File System Services

SERVICE	FUNCTION
Data Transfer Services	
OPEN	Prepare a file for processing.
READ	Transfer data from an open file to memory.
WRITE	Transfer data from memory to an open file.
SEEK	Set or return value of disk file marker.
CLOSE	Terminate processing of an open file.
Directory Maintenance Services	
DELETE	Remove a file from the directory and release its space.
RENAME	Change the name of a file in the directory.
ATTRIB	Change an attribute of a file in the directory.
Other Services	
FORMAT	Initialize a new disk.
LOAD	Read a file of executable code into memory.
DISKIO	Perform basic I/O operations.

The data transfer and directory maintenance services are analogous to the corresponding ISIS-II system calls, and disks processed with these DFS services will be compatible with ISIS-II. The file structure used by ISIS-II and DFS is described in Appendix E. DFS provides two additional capabilities not present in ISIS-II which significantly enhance the utility of the data transfer services:

1. An unlimited number of files can be open concurrently (subject to memory constraints).

2. Multiple tasks may read the same file concurrently.

The DFS LOAD service operates like the ISIS-II loader, except that control is passed to the loaded segment by the user task rather than by the loader. One way the loader can be used is to perform "overlay" processing, that is, to dedicate a memory space to transient routines that are loaded from disk into the memory space as needed.

The FORMAT service labels (names) a disk and creates the directory and other system files needed for ISIS-II and DFS disk processing. Unlike ISIS-II, DFS has no file copying facility in its FORMAT service, so it cannot be used for disk backup. Any drive may be used to format a disk.

A user task may bypass directory processing and perform I/O operations on the disk directly with the DFS DISKIO service. There is very little system overhead involved in this service, and any part of the disk may be read or written. On the other hand, the services that DISKIO provides are very basic. Comparing DISKIO to the directory-based services is somewhat like comparing assembly language to PL/M: power and "efficiency" are traded off for ease of implementation and maintenance. DISKIO is intended to be used primarily in applications that have extremely demanding performance and/or memory constraints.

Facilities for Real-Time Operation. DFS performs most of the activities required to manage real-time disk operations. The user does not have to be concerned with scheduling and coordinating the multiple and sometimes conflicting requests that typify real-time operations. By taking responsibility for these activities, and for the maintenance of system integrity, DFS encourages the user to concentrate on solving the application problem at hand.

In addition to presenting a complex control situation, real-time applications often have demanding performance requirements. DFS helps maximize system throughput by allowing user tasks to overlap disk operations with processor operations and by allowing multiple disk operations to proceed in parallel. For example, files can be "double buffered" so that while DFS is filling (or emptying) one buffer, the user task can be processing the data in the other, reducing the time the task spends waiting for I/O. If a system has more than one disk controller (iSBC 80/10's are limited to one controller), several controllers can be active at the same time, allowing concurrent access to multiple files.

Device Independence. The Disk File System helps to isolate user tasks from the physical characteristics of specific disk drives and controllers. All references to hardware, such as recording density, are specified by the user in two or three tables. This allows user tasks to be written with minimal software consideration of hardware attributes.

Modular Configuration. DFS is not a monolithic system; each user implements a "version" of the system which fits the requirements of the application. If the application has no need for direct access, for example, then the SEEK service is not included in the user's implementation of DFS. While this procedure slightly increases the initial effort involved in setting up a system, it insures that the memory required by DFS is a function of the application — no space is occupied by unused routines.

Use Environment

The Disk File System operates on iSBC 80/20, 80/30, and 80/10 computers. The system functions similarly on the three machines, and many applications developed on one machine can be run on the others, assuming compatible configurations.

A variety of diskette products can be used with DFS: standard-size single-drive and dual-drive units that record in either single or double density, mini-size units that record in single density, and Intel controllers that can handle up to two or up to four drives.

The minimum software environment for the Disk File System is the RMX/80 Nucleus. Other RMX/80 extension tasks may be selected as desired.

Memory Requirements

Disk File System memory requirements vary with the level of support you need from DFS: more files, more controllers and more types of I/O operations generally increase the need for memory. Appendix D contains a complete breakdown of DFS memory requirements that can be used to calculate the memory needed to support a particular application. (Refer to "Configuration Requirements" at the end of this chapter to determine what DFS tasks you need to implement a system with your particular requirements for controllers and services.) Appendix D also provides memory requirements for four sample system configurations, three of which include DFS. The third example illustrates how a minimal DFS directory-based system can be implemented in the iSBC 80/20 on-board ROM.

DFS is unique among RMX/80 extensions in that some of its memory must be controller-addressable RAM. This is because disk transfers are performed by Direct Memory Access through the system bus. Because the RAM on the iSBC 80/20 and iSBC 80/10 boards is not connected to the bus, in 80/20- and 80/10-based systems off-board RAM must be used for certain DFS areas. On the iSBC 80/30, on-board RAM is connected to the bus, so this restriction does not apply. Note also that memory mapped to the Intellec development system via ICE-80 or ICE-85 cannot be substituted for the controller-addressable RAM, since mapped memory is not accessible to the bus.

Hardware Requirements

The following Intel diskette controllers, and Intel diskette systems using these controllers, are supported by DFS:

- iSBC 201 Diskette Controller (1-2 drives)
- iSBC 202 Double-Density Controller (1-4 drives)
- iSBC 204 Diskette Controller (1-4 drives, either standard or mini-size)

iSBC 80/20 and 80/30 systems can use any number and mix of drives and controllers, subject only to the availability of interrupts and memory space. Systems using iSBC 80/10's are limited to a single controller and therefore to a maximum of two or four drives, depending on which controller is selected.

If mini-size diskette drives are used on an iSBC 80/10-based system, an off-board clock must be configured in the system. (See Appendix G.) This is necessary because timed RQWAIT operations are used by DFS to start and stop the drive motor. The clock is also needed in an iSBC 80/10 system if you wish to use the timeout feature, whereby an error code will be returned if the drive has not responded within a set period of time. (See "Timing Variables for Disk File System" in Appendix G.)

Note that disk controller boards must have higher bus priority than CPU boards. (See discussion of bus priority in Appendix F.)

How the Disk File System Operates

The Disk File System is implemented as a set of tasks that run under the RMX/80 Nucleus. User tasks communicate with DFS by sending messages to specific exchanges.

For example, sending a message to the RQOPNX exchange causes the DFS OPEN service to attempt to open the file specified in the request message. If the attempt is successful (i.e., no errors are detected), the OPEN service creates a new exchange to which subsequent messages requesting I/O to the file are sent. If another task opens the same file, DFS sets up a separate exchange for that task. Two tasks can access the same file through a single exchange if the opening task provides the exchange address to the other task. Using this approach is discouraged, however, since either task can change the file's LENGTH and MARKER indicators (see the discussion of LENGTH and MARKER which follows). In most cases it is better for two tasks which need access to the same file to send requests for access to a third task which does the I/O. Requests to READ, WRITE, SEEK, and CLOSE for different files are processed for the most part in parallel (see the discussion of controller exchanges which follows). READ, WRITE, SEEK, and CLOSE requests which are sent to the same exchange are processed serially.

A separate exchange exists for each of the following DFS services: OPEN, RENAME, DELETE, ATTRIB, FORMAT, LOAD, and DISKIO. Requests for these services are queued at the appropriate exchanges and are processed serially first- in first-out (FIFO). Since these types of services are typically requested sporadically, their serial execution does not seriously impair the throughput of most systems. If a series of these requests (each a different type) must be executed in a guaranteed sequence, the requesting task must wait for each individual request to be acknowledged before issuing the next.

Controllers are associated with tasks, and by implication, with exchanges. If two controllers are allocated separate tasks, they can be operated in parallel. If they are allocated the same task, only one controller can be active at a time, and requests to either controller are placed in the same queue and processed FIFO. The user specifies this controller/task relationship based on a throughput/memory tradeoff: separate tasks have higher throughput potential but require more memory than shared tasks.

The paragraphs and figures that follow explain the operation of the individual DFS services. Note that although the diagrams show DFS as a single box, it is actually a collection of tasks.

OPEN and CLOSE Services

Figure 7-1 illustrates the task-exchange relationships that apply to the OPEN service. Tasks requesting a connection with a file send a request message to the DFS-defined OPEN exchange, RQOPNX. DFS returns a response message to a user-defined exchange, here called RESP$EX, and creates an active file request exchange (here called AFR$EX) through which the task and the file may communicate. Files may be opened for read, write, or update.

The CLOSE service is invoked by sending a message to the active file request exchange created when the file was opened. This service severs the task-file connection (shown by dotted lines in figure 7-1) when it is no longer needed.
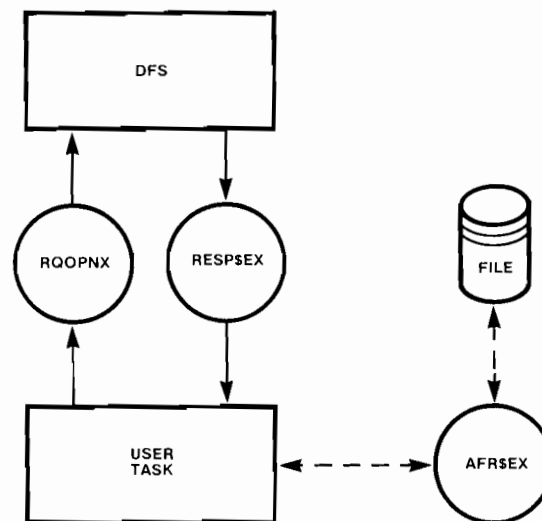
Figure 7-1. Exchanges for OPEN, CLOSE, READ, WRITE, and SEEK

READ, WRITE, and SEEK Services

The task-exchange relationships set up for the READ, WRITE, and SEEK services are the same as shown for OPEN (figure 7-1). After the file is opened, tasks perform these operations by sending request messages to the exchange created by the OPEN exchange (called AFR$EX in the figure). Since each task communicates with the open file through a "private" exchange, requests for I/O may proceed in parallel.

DELETE, RENAME, and ATTRIB Services

The task-exchange relationships set up for the directory services (DELETE, RENAME, and ATTRIB) are illustrated in figure 7-2. The user task sends a request message to the appropriate DFS-defined exchange for the service (RQDELX, RQRNMX, or RQATRX), then waits for a response from DFS at a user-defined response exchange (here called RESP$EX). Using these services, tasks may delete or rename files or change their attributes. As in ISIS-II, file attributes are Invisible, Write Protect, System, and Format.

FORMAT and LOAD Services

Figure 7-3 shows the task-exchange relationships that are used by the FORMAT and LOAD services. A user task may request that a disk be formatted by sending a request message to the DFS-defined exchange RQFMTX. A task may request that a binary file be loaded into memory by sending a request message to the DFS RQLDX exchange. For either command, DFS sends a response message to a user-defined response exchange, called RESP$EX in the figure.

DISKIO Service

Users wishing to bypass the directory-oriented I/O service provided by DFS can access the disk in terms of tracks and sectors using the DISKIO service. All of the basic commands of the disk controller are available through this service. The
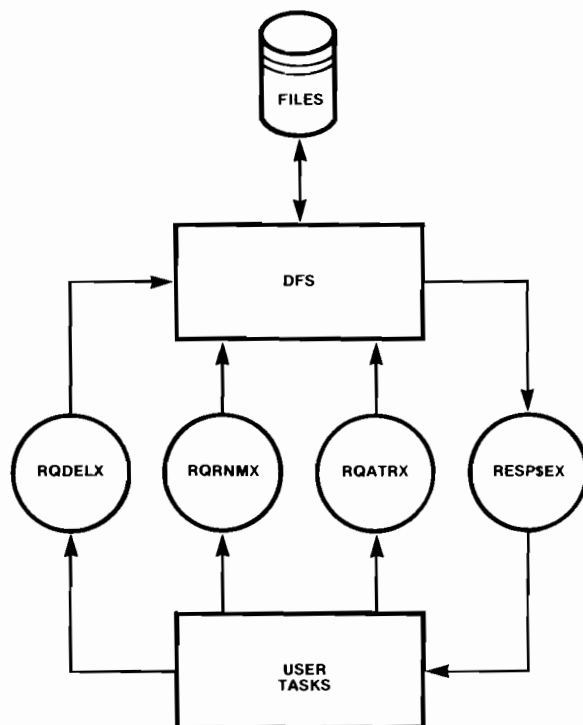
Figure 7-2. Exchanges for DELETE, RENAME, and ATTRIB
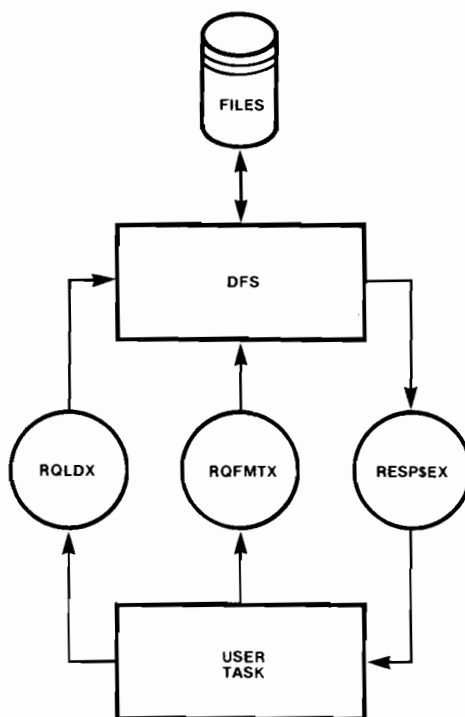
Figure 7-3. Exchanges for FORMAT and LOAD

task-exchange relationships that pertain to DISKIO are shown in figure 7-4. The user task sends a request message to the DFS RQDSKX exchange, and DFS returns a response message to a user-defined exchange, called RESP$EX in the figure.

Using the Disk File System

In this section, two special topics — iSBC 80/10 interrupt polling and motor on/off operations for mini-size diskette drives — are discussed, followed by instructions for making requests for DFS services. At the end of the section, a list of DFS error codes is provided.

iSBC 80/10 Interrupt Polling for DFS

The Disk File System includes interrupt polling routines for the iSBC 80/10 (see Chapter 2, "iSBC 80/10 Interrupts") for disk controllers, so your application does not need to supply these routines. However, each user task in any iSBC 80/10 system must, as part of its initialization, do the following:

*   declare the applicable polling routines EXTERNAL, and

*   associate each with the appropriate interrupt level via an RQSETP operation.

The names of the polling routines (procedure-addresses) are RQHD1V for iSBC 201 and 202 controllers, and RQHD4V for iSBC 204 controllers. The level-number in the RQSETP operation must be the interrupt level assigned to the controller task.
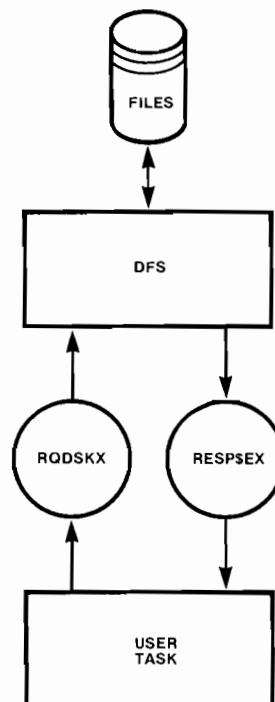
Figure 7-4. Exchanges for DISKIO

Motor On/Off Operations for Mini-Size Drives

If your system includes mini-size diskette drives (supported by the iSBC 204 con-
troller), you must request operations in your application tasks to turn the drive
motor on and off. These operations are provided by the DISKIO service, which is
described later in this chapter.

Before attempting to access any file on a mini-size drive (i.e., before using any other
DFS service that applies to that drive), your system must perform a DISKIO request
specifying MOTOR ON. The MOTOR ON operation includes a one-second time
delay, to ensure the drive motor is up to speed, before it returns. When all accesses
are finished, your system must perform a DISKIO request specifying MOTOR OFF.
The MOTOR ON and MOTOR OFF operations ignore the drive select on the 8271
chip; so if two drives are connected to the same 8271 chip on the controller, both
drives will be affected by a MOTOR ON or MOTOR OFF operation that specifies
either of the drives. (Refer to "Drive Characteristics Table" in the configuration
section of this chapter.)

To conserve the lifetime of your drives, you may perform the MOTOR OFF opera-
tion when you expect there will be no accesses to your mini-size drive (or pair of
drives) for a reasonable period of time, then do another MOTOR ON operation
before the next access request is made. (The length of a "reasonable" period of time
depends upon your application, taking into consideration the tradeoff between drive
wear and the programming overhead of performing MOTOR ON and MOTOR
OFF operations.) You must, however, ensure that all accesses have been completed
before performing a MOTOR OFF. This entails making certain all opened files on
that drive or drive pair have been closed (which implies that all READs, WRITEs,
and SEEKs have been completed), and that responses have been received from DFS
for all requests to the DELETE, RENAME, ATTRIB, FORMAT, LOAD, and
DISKIO services.

An attempt to access a file on a mini-size drive with the motor off will result in a
"Not Ready" I/O error (see "Error Codes" later in this chapter).

Requests for DFS Services

To request any DFS service, the user task must perform the following operations:

* build a message which describes the request.

* send (RQSEND) the message to the appropriate exchange.

* wait (RQWAIT) for the request to be processed.

* check for errors.

Instructions for coding these operations are given in this section. Each DFS service is
covered separately; request message format and rules are provided along with ex-
amples of usage. Several things should be noted about the examples:

* The examples are intended to be succinct illustrations of the essential use of
  each service; they should not be construed as complete, executable user tasks.

* In order to make the usage of the services as clear as possible, the examples
  have been coded in PL/M. The OPEN example, however, has also been coded
  in assembly language to show the corresponding constructs. There is a high
  degree of commonality among the user interfaces to the various DFS services,
  and it should be easy for assembly-language programmers to "translate" the
  other examples as necessary.

* Each example includes an error routine that is empty except for a comment
  which states that user code should replace the comment. This approach has
  been taken in order to emphasize that user tasks must be prepared for errors,
  but that the technique used for error processing is necessarily application-
  dependent.

- Comments show the values that variables (primarily message fields) are expected to contain. In ROM-based applications these values must be moved from ROM constants when the user task is initialized (see "Planning Memory Space" in Chapter 2). RAM areas cannot be initialized in the RMX/80 environment with the PL/M INITIAL or assembly language DB/DW directives unless the data is loaded from an external device.

- User buffer areas may be made controller-addressable or not, depending on several factors which are discussed in the "Defining System Components" section later in this chapter. The examples show how to code both types of buffers.

- The examples show user-coded DFS message formats. PL/M users may include the file DFSMSG.ELT in their programs; this file contains standard declarations of all DFS messages.

LENGTH and MARKER. LENGTH and MARKER are two conceptual entities which are referred to often in the remainder of this chapter. LENGTH is the number of bytes which a file currently contains. Each byte in the file is numbered; the first byte is byte number 0. MARKER is an indication of a task's current position in a file in terms of byte number. READ and WRITE operations transfer data starting at MARKER. If the value of MARKER is 105 and a READ is issued, the first byte transferred to the user task is byte number 105 (the 106th byte in the file). User tasks do not have direct access to the values of LENGTH and MARKER, but these values can easily be calculated when needed by means of the SEEK service. Each task/file combination has its own MARKER; it is not affected by the activities of other tasks that may be reading the same file so long as each task has separately opened the file (i.e., the tasks access the file through different active file request exchanges). LENGTH is not changed by READ operations. If separate tasks are reading the same file through a common exchange (not recommended), both tasks can change MARKER, and precautions must be taken to prevent the tasks from interfering with each other.

File Name Block. Several services (OPEN, ATTRIB, RENAME, DELETE, FORMAT, and LOAD) require one or more file names to be associated with a request for service. These file names are supplied by the user in File Name Blocks, or FNB's. FNB's are generally declared in the task that opens the file, although they may be declared anywhere that is convenient. A field called FILEPTR in the requesting message contains the address of the associated FNB.

Format. The format of the File Name Block is shown in Figure 7-5.
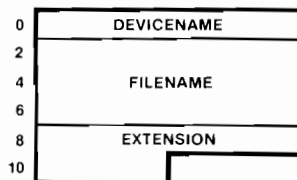
Figure 7-5. File Name Block

DEVICENAME must be two alphanumeric ASCII characters specifying the device upon which the file resides.

FILENAME must be one to six alphanumeric ASCII characters specifying the name of the file. Blanks are not permitted. The characters must be left-justified and any unused positions to the right filled with ASCII nulls (binary zeros).

EXTENSION must be zero to three alphanumeric ASCII characters specifying the extension appended to the file name. Blanks are not permitted. The characters must be left-justified and any unused positions filled with ASCII nulls (binary zeros).

Notes. If the FNB is placed in RAM, the user task must move ROM-based constants to it when the task is initialized by the Nucleus, or load the values from an external device. PL/M users can INCLUDE the file FNB.ELT in their programs to obtain a standard FNB declaration.

Examples. The examples show sample coding for two File Name Blocks. The first file resides on device F1, is named CONTRL and has the extension TXT (in ISIS-II the file would be known as :F1:CONTRL.TXT). The second file is called ZAP1 and resides on unit F2. Note that the unused portions of the FILENAME and EXTENSION fields are filled with binary zeros. Note also that both FNB's are built in ROM; this is the recommended approach unless FNB fields are to be updated by user tasks.

```
/**** FILE NAME BLOCK EXAMPLES (PL/M) ****/

DECLARE        FNB$1        BYTE(11) DATA        ('F1CONTRLTXT');
DECLARE        FNB$2        BYTE(11) DATA        ('F2ZAP1',0,0,0,0,0,);

; FILE NAME BLOCK EXAMPLES (ASSEMBLY LANGUAGE)
               CSEG
FNB1           DB           'F1CONTRLTXT'
FNB2           DB           'F2ZAP1',0,0,0,0,0,
```

OPEN Request. The OPEN service connects a user task and a disk file. The connection is an exchange to which subsequent requests for I/O are directed. So long as memory is available, a task may open an unlimited number of files. The same file may be opened for read-only access by any number of tasks. When a new file is opened for output or update, its attribute bits are reset to zero (see ATTRIB request); the file then cannot be accessed by another task until it is closed, then opened again by the other task.

If a task attempts to READ, WRITE, SEEK or CLOSE a file that is not open to it, the result is unpredictable and can be fatal to the system.

Conflicts can arise in a multitasking environment when more than one task attempts access to the same file. Table 7-2 shows DFS' response when OPEN requests are made under different circumstances. If the open is allowed, the values which LENGTH and MARKER assume are shown; if the operation is disallowed, the error code which is returned in STATUS is shown. "L=current" means that LENGTH is set to the current length of the file.

Table 7-2. Responses of DFS to OPEN Requests

FILE CONDITION AT TIME OF REQUEST	ACCESS REQUESTED		
	READ	WRITE	UPDATE
Does Not Exist	STATUS=13	L=0* M=0	L=0* M=0
Exists Closed	L=current M=0	L=0** M=0	L=current M=0
Open For Read	L=current M=0	STATUS=12	STATUS=12
Open For Write	STATUS=12	STATUS=12	STATUS=12
Open For Update	STATUS=12	STATUS=12	STATUS=12

*Note that the file is created.

**Note that if an existing file is opened for output (WRITE), the file is deleted and re-created empty — any data in the file is lost.

Request Message. The OPEN request message must be sent to the RQOPNX exchange.

The format of the OPEN message is shown in figure 7-6. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.

LENGTH is 17.

TYPE is DFS$OPN (15).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange at which the requesting task waits for a response from DFS. In order to avoid mismatching tasks and messages, a separate response exchange should generally be dedicated to each task/file combination.


```
 0 |         LINK          |
 2 |        LENGTH*        |
 4 |  TYPE*    |
 5 |     HOME EXCHANGE     |
 7 |   RESPONSE EXCHANGE*  |
 9 |        STATUS         |
11 |        FILEPTR*       |
13 |        ACCESS*        |
15 |      AFR EXCHANGE     |
```
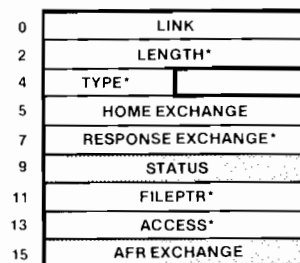
Figure 7-6. OPEN Request Message

When DFS returns this message, it sets STATUS to indicate the results of the OPEN request. Zero is returned if the file is opened successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section later in this chapter. The user must check the contents of STATUS to be sure the file has been opened successfully before attempting I/O operations on the file.

FILEPTR must contain the address of the File Name Block which specifies the file to be opened.

ACCESS must specify one of the following codes: RD$ACC (1), WRT$ACC (2), or UPD$ACC (3). RD$ACC indicates that the file is to be opened for input (reading); WRT$ACC, output (writing); and UPD$ACC, update (both reading and writing). PL/M users may INCLUDE the file OPNACC.ELT in their programs to define the symbolic values for these codes.

When DFS returns the message, it sets AFR EXCHANGE to the address of the Active File Request Exchange that it builds if the file is opened successfully. All READ, WRITE, SEEK, and CLOSE request messages for this task-file combination must be sent to this exchange. If the OPEN is unsuccessful (STATUS ≠ 0), the contents of this field are undefined and should not be accessed by the user task.

Examples. The coding examples open the files defined previously in the File Name Block section: CONTRL.TXT is opened for input (reading) and ZAP1 is opened for output (writing). Message fields that need not be coded by the user are initialized to zero for convenience; this is not required, however.

```
/**** "OPEN" EXAMPLE (PL/M) ****/

/* OPEN MESSAGES */
DECLARE (OPEN$CONTRL$MSG,OPEN$ZAP1$MSG) STRUCTURE
        (LINK                   ADDRESS,
         LENGTH                 ADDRESS,
         TYPE                   BYTE,
         HOME$EXCHANGE          ADDRESS,
         RESPONSE $EXCHANGE     ADDRESS,
         STATUS                 ADDRESS,
         FILEPTR                ADDRESS,
         ACCESS                 ADDRESS,
         AFR$EXCHANGE           ADDRESS);
    /*  ASSUMED CONTENT OF MESSAGES : */
    /*  0, 17, DFS$OPN, 0, .CONTRL$RESPONSE, 0, .FNB1, RD$ACC, 0 */
    /*  0, 17, DFS$OPN, 0, .ZAP1$RESPONSE, 0, .FNB2, WRT$ACC, 0 */

/* OPEN AND RESPONSE EXCHANGES */
DECLARE RQOPNX                  EXCHANGE$DESCRIPTOR EXTERNAL;
DECLARE (CONTROL$RESPONSE,
         ZAP1$RESPONSE)         EXCHANGE$DESCRIPTOR;

DECLARE (T1,ERROR$CODE)         ADDRESS; /* FOR RQWAIT, ERROR ROUTINE */

OPEN$ERROR: PROCEDURE(X);
    DECLARE  X                  ADDRESS;
    /* USER CODE TO HANDLE ERRORS GOES HERE */
    END OPEN$ERROR;
```

```
            /* BUILD RESPONSE EXCHANGES */
            CALL RQCXCH(.CONTRL$RESPONSE);
            CALL RQCXCH(.ZAP1$RESPONSE);

            /* OPEN FILES */
            CALL RQSEND(.RQOPNX,.OPEN$CONTRL$MSG);
            CALL RQSEND(.RQOPNX,.OPEN$ZAP1$MSG);

            /* WAIT UNTIL BOTH OPENS ARE EXECUTED */
            T1 = RQWAIT(.CONTRL$RESPONSE, 0);
            T1 = RQWAIT(.ZAP1$RESPONSE, 0);

            /* CHECK FOR ERRORS */
            IF (ERROR$CODE:=OPEN$CONTRL$MSG.STATUS)<> 0 THEN
              CALL OPEN$ERROR(ERROR$CODE);
            IF (ERROR$CODE:=OPEN$ZAP1$MSG.STATUS)<> 0 THEN
              CALL OPEN$ERROR(ERROR$CODE);
```

```
;"OPEN" EXAMPLE (ASSEMBLY LANGUAGE)
;       CSEG
            EXTRN RQOPNX            ;OPEN EXCHANGE - DEFINED IN CONFIG.MOD.

;BUILD RESPONSE EXCHANGES
            LXI     B,CWAITX       ;ADDR WHERE EXCH IS TO BE BUILT
            CALL    RQCXCH         ;BUILD IT
            LXI     B,ZWAITX       ;ADDR WHERE EXCH IS TO BE BUILT
            CALL    RQCXCH         ;BUILD IT

;OPEN FILES
            LXI     B,RQOPNX       ;ADDRESS OF OPEN EXCHANGE
            LXI     D,CNTMSG       ;MESSAGE DESCRIBING 'CNTL' FILE
            CALL    RQSEND         ;SEND THE OPEN MESSAGE
            LXI     B,RQOPNX       ;ADDRESS OF OPEN EXCHANGE
            LXI     D,ZAPMSG       ;MESSAGE DESCRIBING 'ZAP1' FILE
            CALL    RQSEND         ;SEND THE OPEN MESSAGE

;WAIT UNTIL BOTH OPENS ARE EXECUTED
            LXI     B,CWAITX       ;WHERE WE WANT TO WAIT
            LXI·    D,0            ;NO TIME LIMIT
            CALL    RQWAIT         ;WAIT
            LXI     B,ZWAITX       ;WHERE WE WANT TO WAIT
            LXI     D,0            ;NO TIME LIMIT
            CALL    RQWAIT         ;WAIT

;CHECK FOR ERRORS
CKCNT   EQU     $
NOERR   EQU     0
            LDA     CSTAT          ;LOW BYTE OF STATUS
            CPI     NOERR          ;IS IT ZERO?
            JZ      CKZAP          ;YES, CONTINUE
            LXI     B,CSTAT        ;NO, PASS STATUS TO ERROR RTN
            CALL    OPNERR         ;TAKE CARE OF THE ERROR
CKZAP   EQU     $
            LDA     ZSTAT          ;LOW BYTE OF STATUS
            CPI     NOERR          ;IS IT ZERO?
            JZ      CONTIN         ;YES, FILE OPENED, CONTINUE
            LXI     B,CSTAT        ;NO, PASS STATUS TO ERROR RTN
            CALL    OPNERR         ;TAKE CARE OF ERROR
            JMP     CONTIN         ;GO AROUND ERROR RTN
OPNERR  EQU     $                  ;ERROR ROUTINE

;USER CODE TO HANDLE ERRORS GOES HERE
            RET

CONTIN  EQU     $

            DSEG
```

```
                    ;OPEN MESSAGE FOR 'CNTL' FILE
                    CNTMSG    DS  17              ; VALUES MOVED FROM ROM:
                    CLINK     EQU CNTMSG          ;  0
                    CLEN      EQU CNTMSG+2        ; 17
                    CTYPE     EQU CNTMSG+4        ; 15 (OPEN)
                    CHOMEX    EQU CNTMSG+5        ;  0
                    CRESPX    EQU CNTMSG+7        ; ADDR (CWAITX)
                    CSTAT     EQU CNTMSG+9        ;  0
                    CFILEP    EQU CNTMSG+11       ; ADDR (FNB1)
                    CACCES    EQU CNTMSG+13       ;  1  (READ)
                    CAFRX     EQU CNTMSG+15       ;  0

                    ;OPEN MESSAGE FOR 'ZAP1' FILE
                    ZAPMSG    DS  17              ; VALUES MOVED FROM ROM:
                    ZLINK     EQU ZAPMSG          ;  0
                    ZLEN      EQU ZAPMSG+2        ; 17
                    ZTYPE     EQU ZAPMSG+4        ; 15 (OPEN)
                    ZHOMEX    EQU ZAPMSG+5        ;  0
                    ZRESPX    EQU ZAPMSG+7        ; ADDR (ZWAITX)
                    ZSTAT     EQU ZAPMSG+9        ;  0
                    ZFILEP    EQU ZAPMSG+11       ; ADDR (FNB2)
                    ZACCES    EQU ZAPMSG+13       ;  2 (WRITE)
                    ZAFRX     EQU ZAPMSG+15       ;  0

                    ;LOCAL EXCHANGES
                    CWAITX    DS  10              ; RESPONSE EXCHANGE FOR 'CONTRL'
                    ZWAITX    DS  10              ; RESPONSE EXCHANGE FOR 'ZAP1'
```

READ Request. The READ service transfers data from an open disk file to a user task. The file must be open for input or update. Successive reads to the same file are processed serially in the order in which the requests are made (FIFO). The data transfer begins with the byte at location MARKER and continues until the number of bytes requested by the user has been read or the end of the file is reached, whichever occurs first.

Request Message. The READ request message must be sent to the exchange established by DFS when the file was opened.

The format of the READ message is shown in figure 7-7. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.
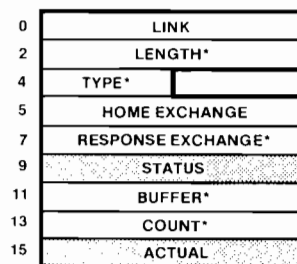
Figure 7-7. READ Request Message

LENGTH is 17.

TYPE is DFS$RD (8).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange at which the requesting task waits for a response from DFS. Although there is no prohibition against setting up a new exchange at which to wait for reads, the straightforward approach is to use the same exchange at which the task waited for the file to be opened. No other task should use this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the READ request. Zero is returned if the data is read successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check STATUS after every READ to ensure that the operation has completed normally.

BUFFER must specify the address of the beginning of the memory area where DFS is to place the requested data. If the buffer is located in the controller-addressable memory module, data can, in some cases, be placed in the buffer by DMA, resulting in a substantial performance improvement. (See "Defining System Components" later in this chapter.) You must ensure that the buffer is large enough to hold the amount of data requested in the COUNT field. If the buffer is short, the adjacent data in memory will be overwritten without indication to the user.

COUNT must specify the number of bytes to be read from the file. If COUNT = 0, the request is processed normally, but no data is read.

When DFS returns this message, it sets ACTUAL to the actual number of bytes read into the user buffer. MARKER is incremented by ACTUAL during each read. ACTUAL can be greater than zero but less than COUNT in two cases: if an I/O error occurs, or if end-of-file is encountered before the requested number of bytes has been transmitted. The condition (STATUS = 0 and ACTUAL = 0) signals that the user is at end-of-file (unless COUNT = 0). Your task should therefore check the contents of ACTUAL as well as STATUS after every read. DFS will continue to return ACTUAL = 0 if you try to read past the end of the file. If ACTUAL is less than COUNT, the data in the user buffer which is beyond ACTUAL should be considered undefined.

Example. In the coding example, the CONTRL.TXT file used in the File Name Block and OPEN examples is read sequentially from beginning to end. A few variables and other identifiers are used which were defined in the previous examples. Fields in the READ message that the user is not required to code are initialized to zero, although this is not strictly necessary. The user buffer is controller-addressable. and is therefore declared EXTERNAL.

```
/**** "READ" EXAMPLE (PL/M) ****/
DECLARE READ$CONTRL$MSG        STRUCTURE(
        LINK                   ADDRESS,
        LENGTH                 ADDRESS,
        TYPE                   BYTE,
        HOME$EXCHANGE          ADDRESS
        RESPONSE$EXCHANGE      ADDRESS,
        STATUS                 ADDRESS,
        BUFFER                 ADDRESS,
        COUNT                  ADDRESS,
        ACTUAL                 ADDRESS);
/*   ASSUMED VALUES: */
/*   0, 17, DFS$RD, 0, .CONTRL$RESPONSE, 0 , .READ$BUF, 256, 0 */
```

```
/* BUFFER FOR DATA DEFINED IN C.A.M. MODULE */
DECLARE READ$BUF (256)                BYTE EXTERNAL;

/* END-OF-FILE CONTROLS */
DECLARE MORE$DATA                 BYTE;
DECLARE TRUE                      LITERALLY '0FFH';
DECLARE FALSE                     LITERALLY '00H' ;

/* ADDRESS OF AFR$EXCHANGE RETURNED BY OPEN */
DECLARE READ$EXCHANGE             ADDRESS;

/* ROUTINE TO HANDLE ERRORS */
READ$ERROR: PROCEDURE(X);
          DECLARE X               ADDRESS;
          /* USER ERROR PROCESSING CODE GOES HERE */
          END READ$ERROR;

/* ASSUME FILE IS OPEN */
/* SET EOF CONTROL AND ADDRESS OF READ EXCHANGE */
MORE$DATA = TRUE;
READ$EXCHANGE = OPEN$CONTRL$MSG.AFR$EXCHANGE;

/* START READING */
DO WHILE MORE$DATA;
    CALL RQSEND(.READ$EXCHANGE ,.READ$ CONTRL$ MSG);

    /* WAIT FOR READ TO COMPLETE */
    T1 = RQWAIT(.CONTRL$ RESPONSE,0) ;

    /* CHECK FOR ERRORS */
    IF (ERROR$CODE:=READ$CONTRL$MSG.STATUS)<>0
        THEN CALL READ$ERROR(ERROR$CODE);

    /*CHECK FOR END OF FILES */
    IF READ$CONTRL$MSG.ACTUAL>0 THEN DO;
        /* CODE TO PROCESS DATA GOES HERE */
        END;
    ELSE MORE$DATA = FALSE; /* STOP READING */
END;
```

WRITE Request. The WRITE service transfers data from user task memory to an open file. The file must be open for output or update. Successive WRITE requests to the same file are processed serially in the order the requests are made. Intermixed READ and WRITE requests to the same file are also processed serially. Data is written on the file starting at location MARKER and continues until the number of bytes requested by the user is transferred.

Request Message. The WRITE request message must be sent to the exchange established by DFS when the file was opened.

The format of the WRITE message is shown in figure 7-8. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.

```
 0  |         LINK         |
 2  |        LENGTH*       |
 4  |  TYPE* |
 5  |     HOME EXCHANGE    |
 7  |  RESPONSE EXCHANGE*  |
 9  |        STATUS        |
11  |        BUFFER*       |
13  |        COUNT*        |
15  |        ACTUAL        |
```
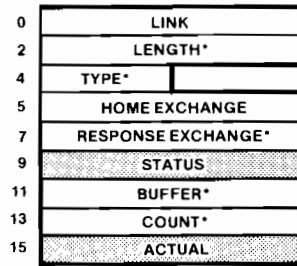
Figure 7-8. WRITE Request Message

LENGTH is 17.

TYPE is DFS$WRT (12).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange at which the requesting task waits for a response from DFS. Although there is no prohibition against setting up a new exchange at which to wait for writes, the straightforward approach is to use the same exchange at which the task waited for the file to be opened. No other task should use this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the WRITE request. Zero is returned if the data is written successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check STATUS after every WRITE to ensure that the operation has completed normally.

BUFFER must specify the address of the beginning of the memory area from which DFS is to write the data. If the buffer is located in the controller-addressable memory module, data can, in some cases, be written from the buffer by DMA, resulting in a substantial performance improvement. (See "Defining System Components" later in this chapter.)

COUNT must specify the number of bytes to be transferred from BUFFER to the file. It is the user's responsibility to ensure that COUNT does not exceed the length of the buffer; if it does, whatever data lies beyond the buffer will be written to the disk without indication to the user. If COUNT = 0, the request is processed normally, but no data is written.

When DFS returns this message, it sets ACTUAL to the actual number of bytes written to the disk. MARKER is incremented by ACTUAL during the write operation. If the write extends the file, LENGTH is also incremented to the new length of the file. If ACTUAL is less than COUNT, an error has occurred. (STATUS identifies the error.)

Example. In the coding example, the file ZAP1 which was used in the File Name Block and OPEN examples is created with ten 80-byte records. Note that if there is already a file called ZAP1 on the disk, its contents are lost. The user buffer is not controller-addressable and is therefore declared and allocated in the module (i.e., it is not declared EXTERNAL).

```
/**** "WRITE" EXAMPLE (PL/M) ****/
/* DEFINE WRITE MESSAGE */
DECLARE WRITE$ZAP1$MSG                STRUCTURE(
        LINK                         ADDRESS,
        LENGTH                       ADDRESS,
        TYPE                         BYTE,
        HOME$EXCHANGE                ADDRESS,
        RESPONSE$EXCHANGE            ADDRESS,
        STATUS                       ADDRESS,
        BUFFER                       ADDRESS,
        COUNT                        ADDRESS,
        ACTUAL                       ADDRESS);
    /*   ASSUMED VALUES: */
    /*   0, 17, DFS$WRT, 0, .ZAP1$RESPONSE, 0 , .WRITE$BUF, 80, 0 /*

DECLARE WRITE$BUF(80) BYTE,           /*   FOR DATA - NO DMA TRANSFER */
        ZAP1$EXCHANGE  ADDRESS,   /*   FILE EXCHANGE */
        I                   BYTE,    /*   LOOP CONTROL */
        T1                  ADDRESS, /*   FOR RQWAIT */
        ERROR$CODE      ADDRESS;

/* HANDLE ANY ERRORS */
WRITE$ERROR: PROCEDURE(X);
        DECLARE X ADDRESS;
        /* USER ERROR HANDLING CODE GOES HERE /*
        END WRITE$ERROR;

/* ASSUME ZAP1 IS OPEN */
/* GET EXCHANGE ADDRESS OF ZAP1 */
ZAP1$EXCHANGE = OPEN$ZAP1$MSG.AFR$EXCHANGE;

/* BUILD THE FILE */
DO I = 1 TO 10;
    /*   CODE TO BUILD RECORD IN WRITE$BUF GOES HERE */

    /*   WRITE THE BUFFER */
    CALL RQSEND(.ZAP1$EXCHANGE,.WRITE$ZAP1$MSG);

    /* WAIT FOR COMPLETION */
    T1 = RQWAIT(.ZAP1$RESPONSE,0);

    /* CHECK FOR ERRORS */
    IF (ERROR$CODE:=WRITE$ZAP$MSG.STATUS) <> 0
        THEN CALL WRITE$ERROR(ERROR$CODE);
END;
```

SEEK Request. The SEEK service gives the user access to the MARKER value associated with an open file. The user may obtain the current value of MARKER or set it to a new value. Since data transfers use MARKER as a starting point, SEEK enables the user to perform direct (or "random") access file processing. Note that SEEK does not transfer data: it merely returns the value of MARKER or sets it to a new value in preparation for subsequent reads and/or writes. Intermixed SEEKs, READs, and WRITEs to the same file are processed serially in the order in which they are requested. SEEK can also be used to extend the LENGTH of a file if the file is open for update.

Some knowledge of the DFS file structure is necessary to use SEEK. While Appendix E contains a complete description of the file structure, the following treatment should suffice for most users.

DFS files are physically organized into blocks of 128 bytes (the length of a disk sector). A standard-size single-density diskette has 1915 of these blocks available for user data. (Some disk space is used by the system for the diskette directory and other system data.) The maximum length of one user file residing on a standard single-density diskette, then, is 1915 x 128 = 245,120 bytes. Standard-size double-density diskettes contain 3,826 blocks or 489,788 bytes; mini-size diskettes, 590 blocks or 75,520 bytes. Note that these maximum figures hold only when there is a single file on the disk; as files are added, additional space is used by the system to keep track of the files, reducing the space available for user data. (See Appendix E for details.) The first block of a file is block number 0; the first byte in a block is byte number 0.

The user controls MARKER with the BLOCKNO (block number) and BYTENO (byte number) fields in the SEEK message. At any time a file is open, MARKER can be expressed in terms of BLOCKNO and BYTENO as follows:

$$MARKER = ((BLOCKNO \times 128) + BYTENO) \ (modulo \ 2^{23})$$

The (modulo 2^{23}) term in the equation does not affect the computation of MARKER unless ((BLOCKNO x 128) + BYTENO) > 8,388,608 — an illegal value for diskette files. Therefore in practice the term can be ignored for diskettes. Notice also that for files that are less than 65,536 bytes in length, BLOCKNO can be set to zero, making MARKER simply equal to BYTENO. For example, in a file 10,000 bytes long, MARKER could be set to 1289 by specifying either of the following:

BLOCKNO = 10	or	BLOCKNO = 0
BYTENO = 9		BYTENO = 1289

However, DFS always sets as high as possible the value of BLOCKNO which it returns to the user. For example, if the user sets MARKER as BLOCKNO = 2, BYTENO = 128 and then requests the current value of MARKER, DFS returns BLOCKNO = 3, BYTENO = 0.

SEEK operates in any of five user-specified modes:

CUR: Return the current value of MARKER.

INCR: Increment MARKER by a specified value (move relatively forward in the file).

DECR: Decrement MARKER by a specified value (move relatively backward in the file).

SET: Set MARKER to a specified value (move to an absolute location in the file).

EOF: Set MARKER to LENGTH and return its value (move to end-of-file).

Table 7-3 illustrates a sequence of legal SEEK requests in a file which is assumed to contain 8692 bytes.

Table 7-3. Responses of DFS to a Sample Sequence of SEEK Requests
(8692-byte file)

Values at Time of Request				Values Following Request			
MARKER	**MODE**	**BLOCKNO**	**BYTENO**	**STATUS**	**MARKER**	**BLOCKNO**	**BYTENO**
0	CUR	—	—	0	0	0	0
0	INCR	10	8	0	1288	10	8
1288	DECR	0	4	0	1284	10	4
1284	SET	22	5	0	2821	22	5
2821	EOF	—	—	0	8692	67	116

The following additional rules apply to SEEK requests:

* The file must be open for input or update.

* Decrementing MARKER past the beginning of the file is not permitted. (An error code will be returned in STATUS.)

* If the file is open for input, attempting to move MARKER beyond the end of the file by SK$SET or SK$INCR is not permitted. (An error code will be returned.)

* If the file is open for update, it can be extended with SK$SET or SK$INCR. This causes LENGTH to be increased. Disk space is not actually allocated for user data, however, until data is written into the extended area. Space for pointer blocks, however, is allocated (see Appendix E). When a read is done on one of these unallocated data areas, ASCII nulls (binary zeros) are placed in the user's buffer.

Request Message. The SEEK request message must be sent to the active file request exchange established by DFS when the file was opened.

The format of the SEEK message is shown in figure 7-9. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.

LENGTH is 17.

TYPE is DFS$SK (13).

HOME EXCHANGE is not used by DFS.

```
 0 |        LINK         |
 2 |       LENGTH*       |
 4 | TYPE*   |           |
 5 |    HOME EXCHANGE    |
 7 | RESPONSE EXCHANGE*  |
 9 |       STATUS        |
11 |        MODE*        |
13 |      BLOCKNO*       |
15 |       BYTENO*       |
```
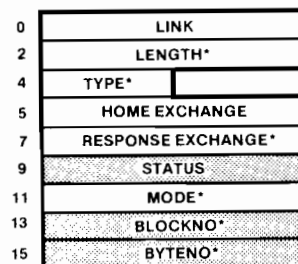
Figure 7-9. SEEK Request Message

RESPONSE EXCHANGE must specify the address of the user-defined exchange at which the requesting task waits for a response from DFS. The simplest technique is to use the same exchange at which the task waited for the file to be opened. No other task should use this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the SEEK request. Zero is returned if the request is processed successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check STATUS after every SEEK to ensure that the operation has completed normally.

MODE must specify one of the following codes: SK$CUR (0), SK$DECR (1), SK$SET (2), SK$INCR (3), or SK$EOF (4). These codes are defined earlier in this section on the SEEK request. PL/M users may INCLUDE the file SEEK.ELT in their programs to define the symbolic values for these codes.

The significance of BLOCKNO depends on the mode specified. If MODE is SK$DECR, SK$SET, or SK$INCR, the BLOCKNO field must specify the block number portion of the MARKER calculation, and DFS does not change this field when it returns the message. If MODE is SK$CUR or SK$EOF, no user coding is required; DFS returns the appropriate value in this field.

The significance of BYTENO depends on the mode specified. If MODE is SK$DECR, SK$SET, or SK$INCR, the BYTENO field must specify the byte number portion of the MARKER calculation, and DFS does not change this field when it returns the message. If MODE is SK$CUR or SK$EOF, no user coding is required; DFS returns the appropriate value in this field.

Example. As an example of using SEEK, assume a file named EXPMNT contains a 100-byte record for each of 40 experiments which are being monitored. Each record begins with an eight-bit temperature field which must be updated periodically. The program shown in the coding example accepts an experiment number (1-40) and a temperature value and updates the corresponding disk record. The entire record is read, although this is not strictly necessary in this case since only the first byte is updated. The user buffer is not controller-addressable.

```
/**** "SEEK" EXAMPLE (PL/M) ****/

UPDATE$TEMP: PROCEDURE(EXPERIMENT,NEW$TEMP);
DECLARE (EXPERIMENT, NEW$TEMP) BYTE;

/* DEFINE SEEK MESSAGE USING SET MODE */

DECLARE  SEEK$MSG                 STRUCTURE(
         LINK                     ADDRESS,
         LENGTH                   ADDRESS,
         TYPE                     BYTE,
         HOME$EXCHANGE            ADDRESS,
         RESPONSE$EXCHANGE        ADDRESS,
         STATUS                   ADDRESS,
         MODE                     ADDRESS,
         BLOCKNO                  ADDRESS,
         BYTENO                   ADDRESS);
    /*   ASSUMED VALUES: */
    /*   0, 17, DFS$SK, 0, .WAIT$EXCH, 0, SK$SET, 0,0 */
```

```
/* RECORD AREA: BUFFER(0) CONTAINS TEMP; DMA TRANSFERS NOT POSSIBLE */
DECLARE   BUFFER(100) BYTE;

/* VARIABLE FOR ADDRESS RETURNED BY RQWAIT */
DECLARE  T1 ADDRESS;

/* ASSUME:
 * EXPMNT HAS BEEN OPENED FOR UPDATE,
 * READ$MSG AND WRITE$MSG HAVE BEEN DECLARED,
 * AFR$EXCHANGE FOR EXPMNT IS EXPMNT$EXCH,
 * TASK WILL WAIT FOR I/O AT WAIT$EXCH.
 */

/* TAKE CARE OF ERRORS */
ERROR$HANDLER: PROCEDURE(X);
    DECLARE  X   ADDRESS;
    /* USER CODE TO PROCESS ERRORS GOES HERE */
    END ERROR$HANDLER;

/* COMPUTE MARKER VALUE - NOTE THAT BLOCKNO IS NOT USED */
SEEK$MSG.BYTENO = (EXPERIMENT −1) * 100;
SEEK$MSG.BLOCKNO = 0;

/* ISSUE SEEK */
CALL RQSEND(.EXPMNT$EXCH,.SEEK$MSG);
T1 = RQWAIT(.WAIT$EXCH, 0);
IF SEEK$MSG.STATUS <> 0 THEN
    CALL ERROR$HANDLER(.SEEK$MSG);

/* READ RECORD INTO BUFFER — NOTE THAT MARKER IS ADVANCED */
CALL RQSEND(.EXPMNT$EXCH,.READ$MSG);
T1 = RQWAIT(.WAIT$EXCH, 0);
IF READ$MSG.STATUS <> 0 THEN
    CALL ERROR$HANDLER(.READ$MSG);

/* UPDATE THE RECORD */
BUFFER(0) = NEW$TEMP;

/* RESET MARKER */
SEEK$MSG.MODE = SK$DECR;
SEEK$MSG.BLOCKNO = 0;
SEEK$MSG.BYTENO = 100;
CALL RQSEND(.EXPMNT$EXCH,.SEEK$MSG);
T1 = RQWAIT(.WAIT$EXCH, 0);
IF SEEK$MSG.STATUS <> 0 THEN
    CALL ERROR$HANDLER(.SEEK$MSG);
SEEK$MSG.MODE = SK$SET;

/* REWRITE RECORD FROM BUFFER */
CALL RQSEND(.EXPMNT$EXCH,.WRITE$MSG);
T1 = RQWAIT(.WAIT$EXCH,0);
IF WRITE$MSG.STATUS <> 0 THEN
    CALL ERROR$HANDLER(.WRITE$MSG);

END UPDATE$TEMP;
```

CLOSE Request. The CLOSE service disconnects a user task from an open disk file. Other tasks reading the same file are not affected so long as the tasks have separately opened the file. DFS processes any outstanding I/O requests before closing the file. When the file is closed, the memory used by DFS to process I/O requests is released, and the exchange which was created when the file was opened is destroyed. If messages are sent to this exchange after the file is closed, the results are unpredictable and may be fatal to the system.

Request Message. The CLOSE request message must be sent to the exchange established by DFS when the file was opened.

The format of the CLOSE message is shown in figure 7-10. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.

0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
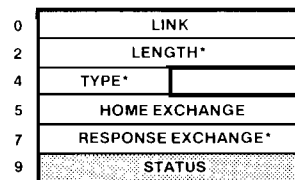
Figure 7-10. CLOSE Request Message

LENGTH is 11.

TYPE is DFS$CLS (14).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange at which the requesting task waits for a response from DFS. The simplest technique is to use the same exchange at which the task waited for the file to be opened. No other task should use this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the CLOSE request. Zero is returned if the request is processed successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check STATUS after performing the CLOSE to ensure that the operation has completed normally.

Example. The following listing provides sample coding of a CLOSE request. This example refers to the coding example given for the OPEN service.

```
/**** "CLOSE" EXAMPLE (PL/M) ****/

/* ASSUME:
            FILE NAMED SAMPLE IS OPEN,
            OPEN MESSAGE IS OPEN$MSG,
            WAIT EXCHANGE IS REPLY$EXCH,
            FILE EXCHANGE IS SAMPLE$EXCH.
```

```
*/
/* FOR ADDRESS RETURNED FROM RQWAIT */
DECLARE T1 ADDRESS;

/* TAKE CARE OF ANY ERRORS */
CLOSE$ERR: PROCEDURE(X);
    DECLARE X ADDRESS;
    /* USER ERROR ROUTINE GOES HERE */
    END CLOSE$ERR;

/* CHANGE OPEN MESSAGE TO CLOSE */
OPEN$MSG.LENGTH = 11;
OPEN$MSG.TYPE = DFS$CLS;

/* CLOSE THE FILE */
CALL RQSEND(.SAMPLE$EXCH,.OPEN$MSG);
T1 = RQWAIT(.REPLY$EXCH,0);
IF OPEN$MSG.STATUS <> 0 THEN
    CALL CLOSE$ERR(OPEN$MSG.STATUS);
```

DELETE Request. The DELETE service removes a file from the disk directory and releases the space allocated to it. Files that are write-protected or open to any task cannot be deleted.

Request Message. The DELETE request message must be sent to the RQDELX exchange.

The format of the DELETE request message is shown in figure 7-11. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.

0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
11	FILEPTR*
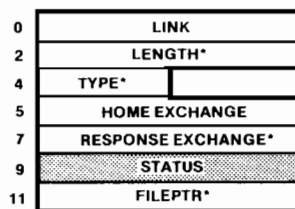
Figure 7-11. DELETE Request Message

LENGTH is 13.

TYPE is DFS$DEL (17).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange at which the requesting task waits for a response from DFS. No other task should use this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the DELETE request. Zero is returned if the request is processed successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check the contents of STATUS after performing the DELETE to ensure that the operation has completed normally.

FILEPTR must provide the address of the File Name Block which specifies the file to be deleted.

Example. The following PL/M code shows how to delete a file.

```
/**** "DELETE" EXAMPLE (PL/M) ****/

/* FILE NAME BLOCK */
DECLARE SUPER$FNB BYTE(11)DATA
                  ('F1SUPER', 0, 'TXT');

/* DELETE MESSAGE */

DECLARE     DEL$MSG              STRUCTURE(
            LINK                 ADDRESS,
            LENGTH               ADDRESS,
            TYPE                 BYTE,
            HOME$EXCHANGE        ADDRESS,
            RESPONSE$EXCHANGE    ADDRESS,
            STATUS               ADDRESS,
            FILEPTR              ADDRESS);
      /*    ASSUMED VALUES: */
      /*    0, 13, DFS$DEL, 0,.WAIT$EXCH, 0,.SUPER$FNB */

DECLARE     WAIT$ADDR            ADDRESS;   /* FOR RQWAIT */

/* DELETE THE FILE */
CALL RQSEND(.RQDELX,.DEL$MSG);
WAIT$ADDR = RQWAIT(.WAIT$EXCH, 0);
IF DEL$MSG.STATUS <> 0 THEN DO;
    /* USER CODE TO HANDLE ERROR GOES HERE */
    END;
```

RENAME Request. The RENAME service changes the name of a file in the directory. Files that are write-protected or open to any task cannot be renamed.

Request Message. The RENAME request message must be sent to the RQRNMX exchange.

The format of the RENAME request message is shown in figure 7-12. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.

```
 0 |            LINK            |
 2 |          LENGTH*           |
 4 |   TYPE*    |               |
 5 |       HOME EXCHANGE        |
 7 |    RESPONSE EXCHANGE*      |
 9 |          STATUS            |
11 |         OLDFILEPTR*        |
13 |         NEWFILEPTR*        |
```

Figure 7-12. RENAME Request Message

LENGTH is 15.

TYPE is DFS$RNM (16).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange where the requesting task waits for a response from DFS. No other task should use this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the RENAME request. Zero is returned if the request is processed successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check STATUS after performing the RENAME to ensure that the operation has completed normally.

OLDFILEPTR must provide the address of the File Name Block that specifies the name of the file which is to be changed.

NEWFILEPTR must provide the address of the File Name Block specifying the new name of the file. The DEVICENAME of both File Name Blocks must be the same. The new name must not be the name of an existing file. (If the name is already in use, an error code will be returned in STATUS.)

Example. The following PL/M code illustrates how to rename a file.

```
/**** "RENAME" EXAMPLE (PL/M) ****/

/* FILE NAME BLOCKS */
DECLARE      OLD$FNB(11)          BYTE DATA
                                  ('F1LAUREL', 0, 0, 0),
             NEW$FNB(11)          BYTE DATA
                                  ('F1HARDY', 0, 0, 0, 0);
```

```
/* RENAME MESSAGE */
DECLARE        RENAME$MSG           STRUCTURE(
               LINK                 ADDRESS,
               LENGTH               ADDRESS,
               TYPE                 BYTE,
               HOME$EXCHANGE        ADDRESS,
               RESPONSE$EXCHANGE ADDRESS,
               STATUS               ADDRESS,
               OLD$FILEPTR          ADDRESS,
               NEW$FILEPTR          ADDRESS);
       /*  ASSUMED VALUES: */
       /*  0, 15, DFS$RNM, 0,.WAIT$EXCH, 0,.OLD$FNB,.NEW$FNB*/

DECLARE        WAIT$ADDR            ADDRESS;  /* FOR RQWAIT */

/* CHANGE LAUREL TO HARDY */
CALL RQSEND(.RQRNMX,.RENAME$MSG);
WAIT$ADDR = RQWAIT(.WAIT$EXCH, 0);
IF RENAME$MSG.STATUS <> 0 THEN DO;
   /* USER ERROR-HANDLING CODE GOES HERE */
   END;
```

ATTRIB Request. The ATTRIB service allows the user to change the attributes of a file. Four attributes are associated with every DFS/ISIS-II file: invisible, system, write-protected and format. Only one of these affects DFS: if a file has the write-protected attribute set (i.e. it cannot be written on), DFS will not open the file for output or update, nor allow it to be renamed or deleted. The other attributes are meaningful in the ISIS-II environment and are described in the ISIS-II System User's Guide, 9800306. All the attributes can be set or reset with the ATTRIB service.

The attributes of a file cannot be changed if the file is open to any task. To create a write-protected file, the sequence of operations is OPEN, WRITE, CLOSE, and ATTRIB.

Request Message. The ATTRIB request message must be sent to the RQATRX exchange.

The format of the ATTRIB message is shown in figure 7-13. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.
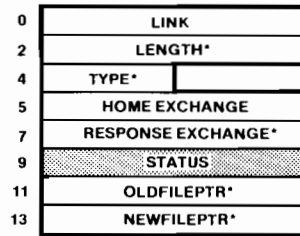
0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
11	FILEPTR*
13	SWID*
15	VALUE*

Figure 7-13. ATTRIB Request Message

LENGTH is 17.

TYPE is DFS$ATR (18).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange at which the requesting task waits for a response from DFS. No other task should use this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the AT-TRIB request. Zero is returned if the request is processed successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check STATUS after performing the ATTRIB operation to ensure that the operation has completed normally.

FILEPTR must provide the address of the File Name Block specifying the file whose attribute is to be changed.

SWID specifies the attribute to be changed, and must be one of the following codes: ATR$INV (0) for Invisible, ATR$SYS (1) for System, ATR$WTP (2) for Write-protect, or ATR$FMT (3) for Format. PL/M users may INCLUDE the file AT-TRIB.ELT in their programs to define the symbolic values for these codes.

The low-order *bit* of VALUE must specify the new value of the attribute identified in SWID. If the low-order bit is 1, the attribute is to be set; if it is 0, the attribute is to be reset.

Example. The following coding example sets the write-protected attribute on a file, thereby making it read-only (until the attribute is reset).

```
/**** "ATTRIB" EXAMPLE (PL/M) ****/

/* FILE NAME BLOCK */
DECLARE VIRGIN$FNB(11)              BYTE DATA
                                   ('F1VIRGIN', 0, 0, 0);

                                   /* ATTRIB MESSAGE */
DECLARE      ATTRIB$MSG            STRUCTURE(
             LINK                 ADDRESS
             LENGTH               ADDRESS,
             TYPE                 BYTE,
             HOME$EXCHANGE        ADDRESS,
             RESPONSE$EXCHANGE    ADDRESS,
             STATUS               ADDRESS,
             FILEPTR              ADDRESS,
             SWID                 ADDRESS,
             VALUE                ADDRESS);
        /*   ASSUMED VALUES: */
        /*   0, 17, DFS$ATR, 0,.WAIT$EXCH, 0,.VIRGIN$FNB,ATR$WTP,0FFFFH */

DECLARE      WAIT$ADDR            ADDRESS;   /* FOR RQWAIT */

/* SEND MESSAGE, WAIT FOR COMPLETION, CHECK FOR ERRORS */
CALL RQSEND(.RQATRX,.ATTRIB$MSG);
WAIT$ADDR = RQWAIT(.WAIT$EXCH, 0);
IF ATTRIB$MSG.STATUS <> 0 THEN DO;
    /* USER ERROR ROUTINE GOES HERE */
    END;
```

LOAD Request. The LOAD service loads a program segment into RAM from a disk file. The loader is a simple non-relocating type that accepts only code that has been "located," that is, assigned absolute memory addresses. The segment must be located so that its memory can be addressed by the controller (i.e., for iSBC 80/20 and 80/10 systems it must be in offboard RAM). Unlike the ISIS-II loader, LOAD does not pass control to the loaded segment, but returns to the user task instead. The service is useful for overlay programming — that is, loading routines that are not used concurrently into a single memory area. Note that only procedures, not main programs, can be loaded. An attempt to execute a main program after loading it will cause the system to malfunction.

Using LOAD successfully requires a thorough understanding of the ISIS-II conventions pertaining to linking and locating object programs. These topics are covered in the ISIS-II System User's Guide, 9800306. Note also that LOAD opens the file that it loads; the user must provide space for a DFS buffer just as though the user task itself were opening the file. (See "Buffers" later in this chapter.)

Request Message. The LOAD request message must be sent to the RQLDX exchange.

The format of the LOAD message is shown in figure 7-14. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.
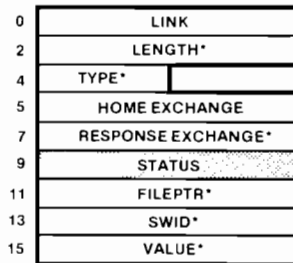
0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
11	FILEPTR*
13	BIAS*
15	ENTRY
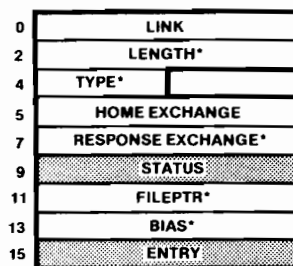
Figure 7-14. LOAD Request Message

LENGTH is 17.

TYPE is DFS$LD (19).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange at which the requesting task waits for a response from DFS. No other task should use this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the LOAD request. Zero is returned if the request is processed successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check the contents of STATUS following completion of the request to ensure that the operation has completed normally.

FILEPTR must provide the address of the File Name Block which specifies the file to be loaded.

BIAS must specify a value which DFS is to add (modulo 64K) to the LOAD address it obtains from the module. This gives the user the ability to place the module into a different (higher) memory location than the address to which it has been located. However, LOAD does *not* adjust the addresses referenced within the module; unless the user changes these addresses, it is unlikely that the module will execute correctly. Therefore, in most cases the user should specify a BIAS of zero.

When DFS returns this message, it sets ENTRY to the entry point address of the loaded module — or, if the module is not a main program, to zero. Since loading a main program has a catastrophic effect on the RMX/80 Nucleus, in practice, zero is always returned in this field. (This feature is provided to maintain compatibility with ISIS-II.)

Loading Tasks. You may wish to use the DFS LOAD service to implement overlays — that is, to load two or more sections of code that are not used concurrently into a single memory area.

If you wish to overlay code for tasks, you must observe certain RMX/80 requirements. Task code in an overlay must first be loaded into memory, then the task must be created by means of an RQCTSK operation. In addition, you must delete the task before loading and creating subsequent tasks in the same memory area. RMX/80 requires that the starting address of the actual code for the task be known and be available when the RQCTSK operation is performed. Particularly in PL/M, this starting address generally will *not* be the beginning of the overlay segment of memory, because constants will precede the code. There are several ways in which this problem may be circumvented.

One method is to declare Static Task Descriptors for all overlayable tasks in your main system (resident) code, then "cross-link" your main system code with your code for all overlays. This cross-linking is a general method for preparing overlays; it may be used for overlays containing tables, procedures, or subroutines rather than tasks. It is accomplished as follows:

1. Compile (or assemble) all modules which are part of the resident portion.
2. Link and locate these modules. The LOCATE program will supply a list of unresolved external references.
3. For each overlay segment:
   a. Compile (or assemble) all modules.
   b. Link, including the PUBLICS of the resident linked and located portion. (Refer to the ISIS-II Systems User's Guide, 9800306.)
   c. Locate the overlay segment at the predefined overlay starting address.
4. Use the LINK program again on the located resident portion of your system, including the PUBLICs of each file created in step 3.

Note that code starting addresses must be declared EXTERNAL in the STD's, and PUBLIC in the overlay modules.

Another method is to have your main system (resident) code build the Static Task Descriptor dynamically in some portion of RAM, and begin the overlay segment with a trivial module containing a single PUBLIC procedure or subroutine that takes no parameters and declares no constants. This procedure is to call another procedure that performs the actual processing for the task. The trivial module must first be linked to the one that does the actual processing. Since the trivial module declares no constants, its starting address will be the starting address of the overlay segment.

You can include code for more than one task in your overlay segment by making your trivial module a series of calls, provided you determine the number of bytes needed for each CALL instruction and add the proper offset to the starting address in the Static Task Descriptor for each task.

A third method is to declare the Static Task Descriptor at the beginning of the overlay segment. Since the STD is in the same module as the task code, it can be coded to include the starting address. Several STD's and the corresponding task code may be included in an overlay segment, since the length of an STD is known (always 17 bytes).

Linking and locating for the second and third methods is accomplished by performing steps 1, 2, and 3 outlined for the first method. Step 4 is not necessary, since the LOCATE output from step 2 should not show any unresolved external references (except, possibly, for unresolved interrupt exchanges). In addition, the trivial module or the module beginning with the Static Task Descriptor(s) must be specified first to the LINK command in step 3b, to ensure that the module is located at the beginning of the overlay.

For any of these methods, you can determine the starting address of your overlay segments by using the MEMORY array generated by the LOCATE program. To do this, you program your resident portion to load overlays into the MEMORY array; then when locating in step 3c, you use the MEMORY segment starting address provided in the LOCATE output from step 2. A second way to determine the overlay starting address is to fix some address that you are sure will be high enough. The former method is more economical in terms of memory space, but the latter method avoids the necessity to relocate the overlays each time you modify the resident portion.

Loading Subroutines. If the code to be loaded does not need to be executed as a task, a somewhat simpler technique may be used. This is illustrated by the example that follows.

Example. The following PL/M code illustrates how LOAD can be used to provide a simple overlay capability.

```
/**** "LOAD" EXAMPLE (PL/M) ****/

/* ASSUME TWO PROCEDURES, ALPHA AND BETA, RESIDE AS
 * OBJECT CODE FILES ON UNIT F1. BOTH HAVE BEEN LOCATED
 * TO THE SAME ABSOLUTE ADDRESS WHICH THE USER HAS DEDICATED
 * TO AN OVERLAY AREA IN OFFBOARD RAM. THIS TASK LOADS AND
 * EXECUTES ONE OR THE OTHER DEPENDING ON THE SETTING OF A
 * VARIABLE CALLED SWITCH.
 */

/* THE PROCEDURES */
ALPHA:          PROCEDURE(P1, P2)       EXTERNAL;
                DECLARE (P1, P2)        ADDRESS;
                END ALPHA;
BETA:           PROCEDURE(P3)           EXTERNAL;
                DECLARE P3              BYTE;
                END BETA;
```

```
/* FILE NAME BLOCKS */
DECLARE      ALPHA$FNB(11)          BYTE DATA
                                    ('F1ALPHA', 0, 'OBJ'),
             BETA$FNB(11)           BYTE DATA
                                    ('F1BETA', 0, 0, 'OBJ');


/* LOAD MESSAGE */
DECLARE      LOAD$MSG               STRUCTURE(
             LINK                   ADDRESS,
             LENGTH                 ADDRESS,
             TYPE                   BYTE,
             HOME$EXCHANGE          ADDRESS,
             RESPONSE$EXCHANGE ADDRESS,
             STATUS                 ADDRESS,
             FILEPTR                ADDRESS,
             BIAS                   ADDRESS,
             ENTRY                  ADDRESS);
      /*   ASSUMED VALUES: */
      /*   0, 17, DFS$LD, 0,.WAIT$EXCH, 0, 0, 0, 0 */

/* MISCELLANEOUS VARIABLES */
DECLARE      (SWITCH, P3)           BYTE,
             (WAIT, P1, P2)         ADDRESS;

/* CODE TO SET SWITCH, P1, P2, P3 GOES HERE */

   IF SWITCH = 1 THEN DO;                    /*   LOAD ALPHA */
      LOAD$MSG.FILEPTR = .ALPHA$FNB;   /*   POINT TO ALPHA */
      CALL RQSEND(.RQLDX,.LOAD$MSG);   /*   SEND LOAD MSG */
      WAIT = RQWAIT(.WAIT$EXCH,0);     /*   WAIT FOR COMPLETION */
      IF LOAD$MSG.STATUS <> 0 THEN DO;  /*   CHECK FOR ERRORS */
         /* USER ERROR ROUTINE GOES HERE */
         END;
      ELSE CALL ALPHA(P1, P2);         /*   EXECUTE ALPHA */
      END;

   ELSE DO;                                  /*   LOAD BETA */
      LOAD$MSG.FILEPTR = .BETA$FNB;    /*   POINT TO BETA */
      CALL RQSEND(.RQLDX,.LOAD$MSG);   /*   SEND LOAD MSG */
      WAIT = RQWAIT(.WAIT$EXCH, 0);    /*   WAIT FOR COMPLETION */
      IF LOAD$MSG.STATUS <> 0 THEN DO;  /*   CHECK FOR ERRORS */
         /* USER ERROR ROUTINE GOES HERE */
         END;
      ELSE CALL BETA(P3);              /*   EXECUTE BETA */
      END;
```

Note that ALPHA and BETA probably refer to PUBLIC variables in the resident portion of the system, and the permanently resident code refers to the EXTERNAL variables ALPHA and BETA. In this case, steps 1 through 4 (given in the "Loading Tasks" section above) must be followed. Note also that ALPHA and BETA can call RQDTSK to create tasks. This implies a fourth technique for loading tasks—i.e., the tasks are created by overlayed code rather than by permanently resident code.

FORMAT Request. The FORMAT service initializes a disk. The disk is labeled, the directory and other system files are created (see Appendix E), and the rest of the disk is overwritten. (The character written is 0C7H for iSBC 201 and 202 controllers, and

0E5H for iSBC 204 controllers.) If the disk contains any user data, this data is destroyed. The diskette created is equivalent to an ISIS-II non-system diskette and may be used on an ISIS-II system.

Request Message. The FORMAT request message must be sent to the RQFMTX exchange.

The format of this message is shown in figure 7-15. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by DFS when it returns the message.

```
0  |          LINK          |
2  |         LENGTH*        |
4  | TYPE*  |
5  |     HOME EXCHANGE      |
7  |   REPONSE EXCHANGE*    |
9  |        STATUS          |
11 |        FILEPTR*        |
```
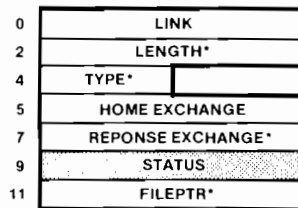
Figure 7-15. FORMAT Request Message

LENGTH is 13.

TYPE is DFS$FMT (20).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange at which the requesting task waits for a response from DFS. No other task should use this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the FORMAT request. Zero is returned if the request is processed successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check the contents of STATUS following completion of the request to ensure that the operation has completed normally.

FILEPTR must contain the address of a File Name Block (FNB). The DEVICENAME field of the FNB specifies the device on which the disk to be formatted resides. The FILENAME and EXTENSION fields of the FNB contain the label to be written on the disk.

Example. The following coding illustrates the use of a FORMAT request.

```
/**** "FORMAT" EXAMPLE (PL/M) ****/

/* FILE NAME BLOCK */
    DECLARE    SENSOR$FNB(11)        BYTE DATA
                                     ('F2SENSORDAT');
```

```
/* FORMAT MESSAGE */
    DECLARE    FMT$MSG              STRUCTURE(
               LINK                 ADDRESS,
               LENGTH               ADDRESS,
               TYPE                 BYTE,
               HOME$EXCHANGE        ADDRESS,
               RESPONSE$EXCHANGE    ADDRESS,
               STATUS               ADDRESS,
               FILEPTR              ADDRESS);
        /*   ASSUMED VALUES: */
        /*   0, 13, DFS$FMT, 0, .WAIT$EX, 0,.SENSOR$FNB */

    DECLARE    WAIT$ADDR            ADDRESS;   /* FOR RQWAIT */

    ABORT:     PROCEDURE(X);        /* IN CASE OF ERROR */
               DECLARE  X           ADDRESS;
               /* USER ERROR ROUTINE GOES HERE */
               END ABORT;

/* FORMAT THE DISK ON DEVICE F2, NAMING IT SENSOR.DAT */
    CALL RQSEND(.RQFMTX,.FMT$MSG);        /*REQUEST SERVICE*/
    WAIT$ADDR = RQWAIT(.WAIT$EX, 0);      /*WAIT*/
    IF FMT$MSG.STATUS <> 0                /*CHECK FOR ERRORS*/
        THEN CALL ABORT(FMT$MSG.STATUS);
```

DISKIO Request. The DISKIO service gives the user the ability to perform disk operations directly, bypassing normal DFS directory processing. Data can be read or written freely anywhere on the disk. DISKIO is intended for applications which have performance and/or memory constraints which preclude the use of the normal directory-based I/O services. No analogous capability exists in ISIS-II.

DISKIO is a powerful facility which has none of the protective features associated with OPEN, READ, WRITE, SEEK and CLOSE. Nothing prevents the user from destroying system files or files in use by other tasks. Therefore a high degree of responsibility is incumbent upon DISKIO users.

Remember also that when using DISKIO the concepts of files, LENGTH, MARKER, and so on do not exist; the user's view of the disk is strictly in terms of tracks and sectors.

Because DISKIO gives the user intimate control of the disk hardware, moderate familiarity with your controller is recommended for successful use of the service. The reader is referred to these Intel publications:
- iSBC 201 Controller: iSBC Diskette Hardware Reference Manual, 9800349
- iSBC 202 Controller: Double-Density Diskette Controller Hardware Reference Manual, 9800420
- iSBC 204 Controller: Single-Density Diskette Controller Hardware Reference Manual, 9800568

The operations which DISKIO can perform are summarized here:

SEEK: Move the head to a specified track.

FORMAT: Format a specified track; except in the case of track 00, the previous track on the diskette must already be formatted. DFS/ISIS-II system files (see Appendix E) are not created by this operation.

RECALIBRATE: Seek track 00.

READ: Transfer the contents of a specified sector to memory.

VERIFY: Validate the Cyclic Redundancy Check characters of a
                   specified sector without transferring data.

WRITE: Transfer data from memory to a specified sector.

WRITE DELETE: Transfer data from memory to a specified sector and mark
                   the sector "deleted." Subsequent READ or VERIFY opera-
                   tions to the sector will result in an I/O error (see "Error
                   Codes" section), but the operations will otherwise be com-
                   pleted normally.

MOTOR ON: (For mini-size drives only.) Turn drive motor on. (For details
                   on the use of this operation, refer to "Motor On/Off Opera-
                   tions for Mini-Size Drives.")

MOTOR OFF: (For mini-size drives only.) Turn drive motor off. (For details
                   on the use of this operation, refer to "Motor/On/Off Opera-
                   tions for Mini-Size Drives.")

Request Message. The DISKIO request message must be sent to the RQDSKX ex-
change.

The format of the DISKIO message is shown in figure 7-16. Asterisks denote fields
that must be coded by the user task before the message is sent. Shaded fields are
those set or changed by DFS when it returns the message.

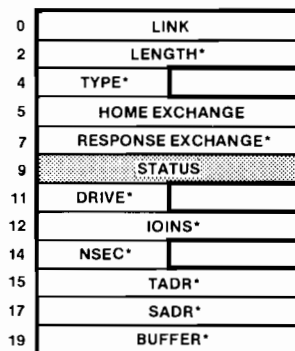| 0 | LINK |
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
11	DRIVE*
12	IOINS*
14	NSEC*
15	TADR*
17	SADR*
19	BUFFER*

Figure 7-16. DISKIO Request Message

LENGTH is 21.

TYPE is DFS$DSK (21).

HOME EXCHANGE is not used by DFS.

RESPONSE EXCHANGE must specify the address of the user-defined exchange at
which the requesting task waits for a response from DFS. No other task should use
this exchange.

When DFS returns this message, it sets STATUS to indicate the result of the DISKIO request. Zero is returned if the request is processed successfully; for details on nonzero values of STATUS, refer to the "Error Codes" section. The user task should check STATUS after each DISKIO request to ensure that the operation has completed normally.

DRIVE must be a number that specifies the disk drive to be accessed by the DISKIO operation. The number coded is the offset (index) of the drive entry in the Device Configuration Table (see "Preparing the Configuration Module"). For example, if the drive to be accessed is the *third* entry in the Device Configuration Table, the number 2 is coded in this field.

IOINS is a code specifying the instruction to be sent to the device controller, as follows: 0 for NO OPERATION; 1 for SEEK; 2 for FORMAT; 3 for RECALIBRATE; 4 for READ; 5 for VERIFY; 6 for WRITE; 7 for WRITE DELETE; 8 for MOTOR ON; or 9 for MOTOR OFF. Codes 10 through 15 are reserved and should not be used.

NSEC must specify the number of 128-byte sectors to be transferred in the request. Permissible values for NSEC are 1-26 for standard single-density diskettes, 1-52 for standard double-density diskettes, and 1-18 for mini-size diskettes. When NSEC is combined with the SADR field, the resulting group of sectors must all be located on the same track. For example, the following request is legal for standard-size double-density diskettes and illegal for standard-size single-density and mini-size diskettes: NSEC = 10, SADR = 20.

TADR must specify the track address for the data transfer. Permissible values are 0-76 for standard-size diskettes and 0-35 for mini-size diskettes.

SADR must specify the starting sector address for the data transfer. Permissible values are 1-26 for standard single-density diskettes, 1-52 for standard double- density diskettes, and 1-18 for mini-size diskettes. Again, when this parameter is combined with NSEC, the sectors to be transferred must all lie on the same track.

BUFFER must specify the starting address of the memory area to which or from which the data is to be transferred. The buffer must be located in controller-addressable memory. (See "Defining System Components" later in this chapter.) All DISKIO transfers are done by DMA. Since DISKIO operations always transfer whole sectors, the length of the buffer should be a multiple of 128. The user task is responsible for ensuring that the buffer length is compatible with the DISKIO request.

If the DISKIO request is FORMAT, the buffer must contain a pair of bytes for each sector on the track for iSBC 201 and 202 controllers, or four bytes for each sector for iSBC 204 controllers. (Consult the appropriate controller Hardware Reference Manual for more detailed coverage of this topic.)

Example. The DISKIO coding example performs the same direct update function that was previously illustrated in the SEEK example. It shows the additional housekeeping functions for which the user is responsible when using the DISKIO service. The data for the experiments is assumed to begin on track 10, sector 01; the disk is standard-size single-density. Note that the entire 100-byte record is read and rewritten. This is not strictly necessary in this example, because only the first byte of the record is updated (only the sector containing the first byte of the record needs to be accessed). It does illustrate, however, what would be required if other fields were to be updated at the same time.

```
/**** "DISKIO" EXAMPLE (PL/M) ****/

UPDATETEMP: PROCEDURE(EXPERIMENT,NEW$TEMP);
 DECLARE (EXPERIMENT,NEW$TEMP)BYTE;

/* DISKIO MESSAGE */
    DECLARE    DISKIO$MSG              STRUCTURE(
               LINK                    ADDRESS
               LENGTH                  ADDRESS,
               TYPE                    BYTE,
               HOME$EXCHANGE           ADDRESS,
               RESPONSE$EXCHANGE       ADDRESS,
               STATUS                  ADDRESS,
               DRIVE                   BYTE,
               IOINS                   ADDRESS,
               NSEC                    BYTE,
               TADR                    ADDRESS,
               SADR                    ADDRESS,
               BUFADR                  ADDRESS);
        /* ASSUMED VALUES: */

        /* 0, 21, DFS$DSK, 0, .WAIT$EX, 0, 0, 0, 0, 0, 0, 0 */


/* BUFFER SPACE FOR TWO SECTORS DEFINED IN C.A.M. MODULE */
    DECLARE     (BUFFER1,BUFFER2) (128)   BYTE EXTERNAL;

    DECLARE    WAIT$ADDR   ADDRESS; /*RQWAIT*/
    DECLARE    /*VARIABLES USED TO COMPUTE DISK ADDRESS OF FIRST BYTE OF RECORD*/

               (TRACKNO,SECTOR,TRACK$OFFSET,
               SECTOR$OFFSET,BYTENO) ADDRESS;

/* TAKE CARE OF ERRORS */
    ERROR$HANDLER: PROCEDURE(X);
        DECLARE X ADDRESS;
        /* USER CODE TO PROCESS ERRORS GOES HERE */
        END ERROR$HANDLER;

/* NOTE THAT THE ENTIRE RECORD IS READ EVEN THOUGH ONLY THE
 * FIRST BYTE IS UPDATED. DRIVE IS DEFINED BY FIRST ENTRY
 * IN DEVICE CONFIGURATION TABLE (NOT DECLARED IN EXAMPLE).
 */

/*   DETERMINE LOCATION OF BEGINNING OF RECORD*/
    BYTENO = ((EXPERIMENT - 1) * 100) + 1;
    TRACK$OFFSET = BYTENO / 128;
    TRACKNO = (TRACK$OFFSET / 26) + 10;
    SECTOR = (TRACK$OFFSET MOD 26) + 1;
    SECTOR$OFFSET = (BYTENO MOD 128) - 1;
```

```
/*   IF RECORD SPANS TWO TRACKS, READ THE FIRST SECTOR OF THE
 *   SECOND OF THOSE TRACKS INTO BUFFER2
 */
IF SECTOR$OFFSET > 28 AND SECTOR = 26 THEN DO;

DISKIO$MSG.IOINS = 4;                          /*DO A READ */
DISKIO$MSG.NSEC = 1;                           /*READ ONE SECTOR */
DISKIO$MSG.TADR = TRACKNO + 1;                 /*DATA IS ON NEXT TRACT */
DISKIO$MSG.SADR = 1;                           /*DATA IN FIRST SECTOR */
DISKIO$MSG.BUFADR = .BUFFER2;                  /*PUT IT IN BUFFER2 */
CALL RQSEND(.RQDSKX,.DISKIO$MSG);  /*SEND MSG */
WAIT$ADDR = RQWAIT(.WAIT$EX,0);    /*WAIT UNTIL DONE */
IF STATUS <> 0 THEN                            /*CHECK FOR ERRORS */
    CALL ERROR$HANDLER(.DISKIO$MSG);
END;


/* READ SECTOR CONTAINING BEGINNING OF RECORD INTO BUFFER1.
 * IF RECORD SPANS TWO SECTORS ON SAME TRACK, READ THE SECOND
 * SECTOR INTO BUFFER2 WITH THE SAME REQUEST
 */

IF SECTOR$OFFSET > 28 AND SECTOR < 26
    THEN DISKIO$MSG.NSEC = 2
    ELSE DISKIO$MSG.NSEC = 1;
DISKIO$MSG.IOINS = 4;
DISKIO$MSG.TADR = TRACKNO;
DISKIO$MSG.SADR = SECTOR;
DISKIO$MSG.BUFADR = .BUFFER1;
CALL RQSEND(.RQDSKX,.DISKIO$MSG);
WAIT$ADDR = RQWAIT(.WAIT$EX,0);
IF STATUS <> 0 THEN
    CALL ERROR$HANDLER(.DISKIO$MSG);

/* UPDATE THE RECORD */
  BUFFER(SECTOR$OFFSET) = NEW$TEMP;


/* REWRITE BUFFER1 (AND BUFFER2 IF IT IS ON THE SAME TRACK) */
  DISKIO$MSG.IOINS = 6;  /*NSEC,TADR,SADR ARE STILL SET*/
  CALL RQSEND(.RQDSKX,.DISKIO$MSG);
  WAIT$ADDR = RQWAIT(.WAIT$EX,0);
  IF STATUS <> 0 THEN
      CALL ERROR$HANDLER(.DISKIO$MSG);


/* REWRITE BUFFER2 IF IT IS ON A DIFFERENT TRACK */
  IF SECTOR$OFFSET > 28 AND SECTOR = 26 THEN DO;
  DISKIO$MSG.TADR = TRACKNO + 1;
  DISKIO$MSG.SADR = 1;
  DISKIO$MSG.BUFADR = .BUFFER2;
  CALL RQSEND(.RQDSKX,.DISKIO$MSG);
  WAIT$ADDR = RQWAIT(.WAIT$EX,0);
  IF STATUS <> 0 THEN
      CALL ERROR$HANDLER(.DISKIO$MSG);
  END;
END UPDATE$TEMP;
```

Error Codes

As a part of processing each user request, DFS places a code in the STATUS field of the user's request message which indicates the result of the request. This code is placed in the low-order byte of STATUS. Your application task should always check this code before processing any data associated with the request.

Table 7-4 lists the error codes that may be found in the low-order byte of STATUS. These error codes can be considered to be an extension of the codes associated with ISIS-II error messages. Most ISIS-II error codes are applicable to problems that can occur in DFS and are retained. Those that are not applicable are deleted, although the number associated with each such number is not reassigned. New error codes that are unique to DFS are added after the last ISIS-II code. Among those errors that are common to both systems, there is one major change: errors that are considered fatal in ISIS are not so considered in DFS.

If error 24 (input/output error) occurs, DFS places one (or more, if multiple errors occur) codes in the *high-order* byte of STATUS to identify the type of I/O error. The I/O error codes that may be found in the high-order byte of STATUS are listed in table 7-5. These codes correspond to the error codes supplied by the disk controller.

Consult the appropriate Hardware Reference Manual for more information on I/O errors. Note that when an I/O error occurs, DFS retries the operation three times before returning an error indication; therefore there is no point in the user task's retrying the operation.

Table 7-6 indicates which types of errors can occur for each type of DFS service request.

Table 7-4. DFS Error Codes (Low-Order Byte of STATUS)

CODE	MEANING
0	No error detected
4	Illegal FILENAME specified in File Name Block
5	DEVICENAME in File Name Block not in Device Configuration Table
6	Attempt to write to a file opened for input
7	No more space on disk
8	Attempt to read a file opened for output
9	No more room in disk directory
10	File Name Blocks in RENAME request do not specify same device
11	Cannot rename file; name already in use
12	File already open
13	No such file
14	Attempt to open a write-protected file for output or update, or attempt to delete or rename a write-protected file
16	Incorrect object program format
18	Unrecognized message TYPE

Table 7-4. DFS Error Codes (Low-Order Byte of STATUS) - (continued)

CODE	MEANING
20	Attempt to seek backward past beginning of file
22	Illegal ACCESS in OPEN message
24	Input/output error on disk
26	Illegal SWID in ATTRIB message
27	Illegal MODE in SEEK message
30	Drive not ready
31	Attempt to seek on file open for output
32	Attempt to delete an open file
35	Attempt to seek past end of file opened for input
40	Request sent to wrong exchange
41	Insufficient free memory space to open file
42	DRIVE specified in DISKIO request is not in Device Configuration Table
43	Drive timeout - the drive has not responded to an I/O request within a set period of time (10 seconds for iSBC 80/20 or 80/30 systems; for 80/10 systems, refer to Appendix G)
44	SEEK request with SEEK not present in system
45	Format driver missing

Table 7-5. DFS I/O Error Codes (High Order Byte of Status)

CODE	MEANING
1H	Deleted record
2H	Cyclic Redundancy Check character error (data field)
3H	Invalid address mark
4H	Seek error
8H	Address error
0AH	Cyclic Redundancy Check character error (ID field)
0EH	No address mark
0FH	Incorrect data address mark
10H	Data overrun or underrun
20H	Write protect
40H	Write error
80H	Not ready

Table 7-6. Disk File System Error Matrix

CODE	OPEN	READ	WRITE	SEEK	CLOSE	DELETE	RENAME	ATTRIB	FORMAT	LOAD	DISKIO
4	X					X	X	X	X	X	
5	X					X	X	X	X	X	
6			X								
7	X		X	X							
8		X									
9	X										
10							X				
11							X				
12	X										
13	X					X	X	X		X	
14	X					X	X				
16										X	
18*											
20				X							
22	X										
24	X	X	X	X	X	X	X	X	X	X	X
26								X			
27				X							
30	X	X	X	X	X	X	X	X	X	X	X
31				X							
32						X		.			
35				X							
40	X	X	X	X	X	X	X	X	X	X	X
41	X										
42											X
43	X	X	X	X	X	X	X	X	X	X	X
44				X							
45									X		

* Since OPEN, READ, etc. represent valid message TYPES, error 18 cannot occur for valid DFS requests.

Configuration, Linking, and Locating

This section supplies the information you need to include Disk File System services in your configuration module, and to link and locate the resulting object code. General instructions for preparing the configuration module, linking, and locating are given in Chapter 3.

Because the Disk File System requires several additional items of configuration information besides the task and exchange information required for other RMX/80 extensions, this section is written to parallel the "Generating the Software Configuration" section in Chapter 3. Under "Preparing the Configuration Module" in the following section, user instructions are provided for the set of additional assembly language macros provided (in RMXEXC.MAC and DFSCFG.MAC) for use in coding configuration modules that include DFS.

In addition to the configuration module, an additional module — the controller-addressable memory module — must be coded and linked into your system if you are using an iSBC 80/10 or 80/20. Instructions for preparing this module are provided following "Preparing the Configuration Module."

At the end of this section are linking and locating instructions for systems that include DFS.

An application system that includes DFS is used as an example throughout this section. The key characteristics of this application are:

- DFS services used: OPEN, CLOSE, READ, WRITE, SEEK
- User tasks: two, called UTASK1 and UTASK2
- Other RMX/80 tasks: none
- Disk configuration: one iSBC 201 controller, two drives
  two iSBC 204 controllers, one with one standard-size drive
  and one with two mini-size drives

Defining System Components

For an application system that includes DFS services, configuration information must be supplied to answer all of these questions:

- What tasks are to be included in the system, and what are the characteristics of these tasks (priorities, etc,)?

- What exchanges are to be defined when the system is initialized?

- What PUBLIC variables (if any) need to be defined for use by RMX/80 extension tasks?

- What are the characteristics of each disk controller in the system?

- What are the characteristics of each disk drive in the system?

- How are Disk File System buffers to be allocated?

This section provides System Definition Worksheet entries to answer these questions for the example application. Part I of the Worksheet (figure 3-1) covers tasks, exchanges, and PUBLIC variables; Part II (figure 7-17) includes controllers, drives, and buffers.

Tasks. In addition to user tasks and tasks for other RMX/80 extensions, two types of DFS-supplied tasks must be included in systems using DFS: DFS service tasks and controller tasks.

DFS Service Tasks. At this point you should know which DFS services are required for your application. The tasks that must be included in your configuration module to implement these services can be determined from figure 7-18, which illustrates DFS service/task relationships.

With the exception of SEEK, which is covered in the next paragraph, each block in the diagram represents a task. Each block also represents a DFS service, except that the DIRSVC (directory services) task includes four services. The diagram also shows how some tasks require other tasks as prerequisites. For example, if a particular application needs the LOAD service, three tasks must be entered on the worksheet: LOAD, DIRSVC and DISKIO.

SEEK is an exception to this scheme. While SEEK requires the DIRSVC and DISKIO tasks, as the diagram shows, it is not considered a task and is not entered on the worksheet nor coded in the configuration module. Instead, SEEK is added into the system via a link parameter; this will be discussed when the general topic of linking the system is undertaken.

One final wrinkle: if the user selects dynamic buffer allocation (discussed later), the RMX/80 Free Space Manager is an additional task which must be entered on the worksheet.

RMX/80 SYSTEM DEFINITION WORKSHEET
(PART II)

4. CONTROLLERS:

NUMBER	TYPE	BASE ADDR	INTERRUPT LEVEL	INTERRUPT EXCHANGE	REQUEST EXCHANGE

5. DRIVES:

DEVICE NAME	TYPE	CONTROLLER NUMBER	UNIT

DEVICE NAME	TYPE	CONTROLLER NUMBER	UNIT

6. DRIVE PARAMETERS (ISBC 204 CONTROLLERS)

CONTROLLER BASE ADDR	CHIP SELECT	STEP RATE	HEAD SETTLING TIME	INDEX COUNT/ LOAD TIME

7. BUFFERS:

A. DFS: ☐ STATIC (_____FILES) ☐ DYNAMIC ☐ NONE

B. USER:____ CONTROLLER ADDRESSABLE ____ NOT CONTROLLER ADDRESSABLE   ☐ NONE
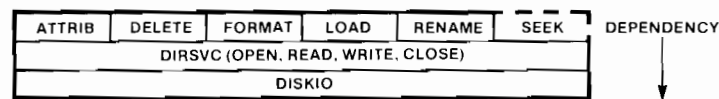
Figure 7-17. RMX/80 System Definition Worksheet, Part II

ATTRIB	DELETE	FORMAT	LOAD	RENAME	SEEK	DEPENDENCY
DIRSVC (OPEN, READ, WRITE, CLOSE)						
DISKIO						

Figure 7-18. DFS Service/Task Relationships

Controller Tasks. The user assigns each disk controller to a DFS-supplied task which coordinates the operation of the controller. In a system with multiple controllers, each controller may be given its own task, all controllers may share a single task, or there may be a combination of shared and exclusive controller tasks. iSBC 201 and 202 controllers, however, cannot share a task with iSBC 204 controllers. (Recall that iSBC 80/10 systems are limited to a single controller.) If two controllers have exclusive tasks, the controllers may be operated concurrently. (For example, controller 1 can be reading while controller 2 is writing.) If two controllers share a task, however, they cannot be active concurrently. (If controller 1 is reading, controller 2 must wait until controller 1 is finished before it can do anything.) In short, exclusive tasks enable parallel controller operations, while shared tasks necessitate serial controller operation.

How controllers are assigned to tasks depends on the characteristics of the application and the result of a memory/throughput tradeoff. If the application has no logical opportunity for parallel controller operation (i.e., there is no situation where two controllers can be active at the same time), then there is no point in creating separate controller tasks. If there is a logical opportunity for parallel operation, and performance requirements make it desirable to exploit this opportunity, then separate controller tasks should be considered. The potential for increased throughput must be weighed against the increased memory (see Appendix D) which separate controller tasks demand. Since the memory required for controller tasks is comparatively small, the tradeoff will usually indicate exclusive tasks for all controllers which can use them to advantage, except in applications that are extremely memory-constrained.

Each controller must have its own interrupt level — i.e., no two controllers can share an interrupt level — and the corresponding interrupt exchange must be declared in your system.

Task Worksheet Entries. Table 7-7 contains standard worksheet entries for DFS service and controller tasks. Omitted (dashed) entries are to be supplied by the user.

Note that all iSBC 201 and 202 controller tasks in your system, and likewise all iSBC 204 controller tasks, will have the same Initial Program Counter. You create separate controller tasks for the same type of controller by assigning a different NAME to each task. This is an example of the principle discussed in "Code Shared by More than One Task" in Chapter 3; RQHD1 and RQHD4 are reentrant modules supplied by DFS.

The Default Exchange is optional for user tasks, but must be supplied for all DFS service and controller tasks. You must name your own default exchanges for controller tasks.

Table 7-7. Configuration Information for DFS Service and Controller Tasks

TASK	NAME	INITIAL P.C.	STACK LENGTH	PRIORITY	DEFAULT EXCHANGE
DISKIO	'DISKIO'	RQPDSK	48	—	RQDSKX
Directory Services	'DIRSVC'	RQPDIR	48	—	RQDIRX
ATTRIB	'ATTRIB'	RQPATR	64	—	RQATRX
DELETE	'DELETE'	RQPDEL	64	—	RQDELX
FORMAT	'FORMAT'	RQPFMT	64	—	RQFMTX
LOAD	'LOAD '	RQPLD	64	—	RQLDX
RENAME	'RENAME'	RQPRNM	64	—	RQRNMX
ISBC 201 and 202 Controllers	—	RQHD1	80	—	—
iSBC 204 Controllers	—	RQHD4	80	—	—

Assigning Priority Levels. Refer to "Assigning Priority Levels" in the configuration section of Chapter 3 for general suggestions on this subject. In addition, certain rules and guidelines apply to assigning priorities to DFS service and controller tasks.

In general, the only DFS tasks which should be assigned priorities numerically lower than 129 (those priorities corresponding to interrupts) are controller tasks. Each of these tasks should be given a software priority corresponding to the hardware interrupt level which has been assigned to the controller. Generally, high-speed devices such as disk controllers are run at relatively high (numerically low) interrupt levels such as 2 or 3. The software priorities which correspond to these interrupt levels are 33 through 48 and 49 through 64, respectively.

DFS service tasks should be assigned priorities in the range 129-254. Within this range DISKIO should be given the highest priority, and other tasks should be given descending priorities based on their performance characteristics.

Initial Exchanges. You must include the following initial exchanges:

- The standard default exchange (from table 7-7) required for each DFS service task you include in your application. (You must declare these exchanges EXTERNAL.)

- A user-defined default exchange for each controller task you define.

- An interrupt exchange (declared PUBLIC) for each interrupt level used by a controller task.

- Exchanges required by your application tasks and by other RMX/80 extension tasks you are using (including the Free Space Manager if dynamic buffer allocation is selected).

PUBLIC Data. For iSBC 80/20- and 80/30-based systems, the only PUBLIC data item that need be defined (excluding the tables, which are covered elsewhere on the worksheet) is RQNDEV, the number of devices (drives) in the system. In PL/M, this number is generally defined in coding the Device Configuration Table (discussed later in this chapter). In using assembly-language macros, it is defined implicitly by the number of references to the DCT macro (see "Assembly-Language Macros for DFS" later in this chapter).

For iSBC 80/10-based systems, two additional PUBLIC data items must be defined: RQTOV and RQMOTM. These are discussed under "Timing Constants for Disk File System" in Appendix G.

Controllers. As shown on Part II of the worksheet, six items must be entered for each controller in your system: number, type, base address, interrupt exchange, interrupt level, and request exchange.

NUMBER is an arbitrary controller identifier used to associate drive information with controller information. The "first" controller in a system is number 0.

TYPE is (iSBC) 201, 202, or 204.

The controller's BASE ADDRESS is the value set in a switch on the controller board.

INTERRUPT EXCHANGE is in the form "RQLnEX," where "n" is the number of the interrupt level assigned to the controller.

INTERRUPT LEVEL is also set via an on-board switch.

REQUEST EXCHANGE is the same as the Default Exchange specified for the controller task associated with the controller.

Drives. The following items of information must be supplied on the worksheet for each disk drive in your system: device name, type, controller number, and unit.

DEVICE NAME is the logical identifier used to address the drive in File Name Blocks. It is in the form of any two alphanumeric ASCII characters.

TYPE is the type of controller to which the drive is physically attached. For iSBC 204 controllers, enter 204(s) for standard-size or 204 (m) for mini-size drives.

CONTROLLER NUMBER logically links the drive to the particular controller to which it is attached.

UNIT is the number of the drive as "seen" by its controller. For drives attached to iSBC 201's, UNIT may be 0 or 1. For iSBC 202's and 204's, UNIT may be 0, 1, 2, or 3. Each CONTROLLER NUMBER/UNIT combination must be unique.

Drive Parameters (iSBC 204 Controllers). This worksheet information is required only for systems using the iSBC 204 controller. It supplies the parameters required to program the Intel 8271 Floppy Disk Controller chips on the controller. Each controller is supplied with one 8271 chip; the chip can control one or two drives. If a single chip controls two drives, both drives must have the same parameters; however, an additional 8271 can be installed by the user and programmed with different parameters. One table entry is needed for each 8271 chip you use in your system.

Note that although two 8271 chips on an iSBC 204 controller may be programmed with different parameters, a single controller cannot, under RMX/80, operate both standard-size and mini-size drives. If your system uses both types of drives, separate controllers are needed.

CONTROLLER BASE ADDRESS is the iSBC 80 I/O port address that is the base address for the controller. (Although it is called an "address", this parameter is only one byte long.)

CHIP SELECT indicates which of the two possible Intel 8271 chips on the iSBC 204 controller is being programmed. Its value is 0 for the first chip (standard), or 1 for the second chip (user-installed).

STEP RATE, HEAD SETTLING TIME, and INDEX COUNT/LOAD TIME are the drive characteristics parameters. The range and significance of the values for these parameters are described in the Intel iSBC 204 Single-Density Diskette Controller Hardware Reference Manual, 9800568.

Buffers. If the user opens a file or uses the LOAD service (which itself opens a file), then buffer space must be made available to DFS.

There are two ways to allocate this space: statically and dynamically. In static buffer allocation, the user specifies the maximum number of files that will ever be open at the same time and supplies a controller-addressable RAM area which DFS manages as a buffer pool. In dynamic memory allocation, the user includes the Free Space Manager in the system, and DFS acquires and releases buffer space through the Free Space Manager.

The Free Space Manager itself takes a certain amount of memory (see Appendix E), and the user must decide whether this "overhead" is justified in the application. If the Free Space Manager is needed in the application for purposes other than DFS buffer allocation, then dynamic buffer allocation should generally be specified on the worksheet. No extra space, and very little time, is required for the Free Space Manager to manage buffer allocation. Depending on the characteristics of the application, when files are closed the Free Space Manager may be able to use the released space for other purposes.

Note that if dynamic buffer allocation is selected, the user must ensure that DFS always receives memory which is controller-addressable. As discussed under "Memory Requirements" at the beginning of this chapter, in iSBC 80/20- and 80/10-based systems, only off-board RAM is controller-addressable. In 80/20 or 80/10 systems, your tasks should therefore give the Free Space Manager only off-board memory when it is initialized. On-board memory may be given to the Free Space Manager only if the user can ensure that it will never be allocated to DFS. (If the amount of contiguous on-board memory is less than what is needed to open a file — see Appendix D — this will be true.) These restrictions do not apply to iSBC 80/30-based systems, since 80/30 on-board RAM is controller-addressable.

If the Free Space Manager is not otherwise needed in the application, then static buffer allocation should usually be specified. The maximum number of concurrently open files should be noted on the worksheet (remember that LOAD opens a file).

If no files are ever opened or loaded, then no DFS buffers are required, and "none" should be checked on the worksheet.

The following DFS services require user buffers: READ, WRITE and DISKIO (the address of the user buffer is contained in the BUFFER field of the requesting message). In the case of READ and WRITE, a user buffer must be supplied in addition to the buffer needed by DFS to open the file. The question to be answered on the worksheet is whether these buffers need to be located in controller-addressable memory. This in turn is dictated by whether the controller will ever do a Direct Memory Access of the buffer. In the case of DISKIO requests this is always true, so for iSBC 80/10 and 80/20 systems, DISKIO buffer space must always be located off-board.

In the case of READ and WRITE, the rule is as follows: a DMA transfer to/from the user buffer occurs whenever MARKER is at a sector boundary and the request is for 128 bytes or more. Another way of stating this is that DMA transfers are done in whole sectors. If a DMA transfer can occur, then the user buffer must be located in controller-addressable memory. If a DMA transfer cannot occur (e.g., if transfers are always less than 128 bytes), then the user buffer in 80/20 or 80/10 systems can be located on- or off-board (access is faster if it is on-board). It is possible to locate

some user buffers on-board and some off-board. Note that DMA data transfers are significantly faster than if the processor places the data in the user's buffer one byte at a time. If the application is designed to use DMA, throughput is maximized.

Worksheet Entries for Example Application. Figure 7-19 shows how the two parts of the System Definition Worksheet might be completed for our example application.

RMX/80 SYSTEM DEFINITION WORKSHEET
(PART I)

1. TASKS:

2. INITIAL EXCHANGES:

NAME	INITIAL P.C.	STACK LENGTH	PRIORITY	DEFAULT EXCHANGE	NAME
DISKIO	RQPDSK	48	129	RQDSKX	RQDSKX
DIRSVC	RQRDIR	48	135	RQDIRX	RQDIRX
CNTRL1	RQHD1	80	33	CNTL1X	CNTL1X
CNTRL2	RQHD4	80	49	CNTL2X	CNTL2X
CNTRL3	RQHD4	80	66	CNTL3X	CNTL3X
UTASK1	UTASK1	64	150	UTSK1X	UTSK1X
UTASK2	UTASK2	80	160	UTSK2X	UTSK2X
					RQL2EX
					RQL3EX
					RQL4EX

3. PUBLIC DATA:

SYMBOL	VALUE	SYMBOL	VALUE	SYMBOL	VALUE
RQNDEV	5				

**Figure 7-19. Completed System Definition Worksheet for Example Application
(Sheet 1 of 2)**

RMX/80 SYSTEM DEFINITION WORKSHEET
(PART II)

4. CONTROLLERS:

NUMBER	TYPE	BASE ADDR	INTERRUPT LEVEL	INTERRUPT EXCHANGE	REQUEST EXCHANGE
O	201	88 H	2	RQL2EX	CNTL 1X
1	204	70H	3	RQL3EX	CNTL 2X
2	204	60H	4	RQL4EX	CNTL 3X

5. DRIVES:

DEVICE NAME	TYPE	CONTROLLER NUMBER	UNIT
SA	201	O	O
sB	201	O	1
SC	204(s)	1	O
M1	204(m)	2	O
M2	204(m)	2	1

DEVICE NAME	TYPE	CONTROLLER NUMBER	UNIT

6. DRIVE PARAMETERS (ISBC 204 CONTROLLERS)

CONTROLLER BASE ADDR	CHIP SELECT	STEP RATE	HEAD SETTLING TIME	INDEX COUNT/ LOAD TIME
70H	O	8 ms	10 ms	10/35 ms
60H	O	40 ms	20 ms	10/80 ms

7. BUFFERS:

    A. DFS: ☒ STATIC (__3__ FILES) ☐ DYNAMIC ☐ NONE

    B. USER: _1_ CONTROLLER ADDRESSABLE _2_ NOT CONTROLLER ADDRESSABLE ☐ NONE

Figure 7-19. Completed System Definition Worksheet for Example Application
(Sheet 2 of 2)

Recall that the DFS services needed by this application are OPEN, READ, WRITE, SEEK, and CLOSE. All of these services except SEEK are contained in the DIRSVC task. (SEEK will be included in the system at link time.) DIRSVC uses DISKIO, so this task is also included on the worksheet.

Each disk controller is to have its own separate task: CNTRL1 for the iSBC 201 controller, CNTRL2 for the first iSBC 204 controller, and CNTRL3 for the second iSBC 204. Note that CNTRL2 and CNTRL3 specify the same Initial Program

Counter, RQHD4. The three controllers are assigned to interrupt levels 2, 3, and 4, respectively, so their controller tasks are assigned priorities corresponding to these interrupts. Note that the controller tasks have the highest priorities in the system, with DISKIO and DIRSVC following.

The EXCHANGES portion of the worksheet lists the default exchanges for the DFS service and controller tasks, and the three interrupt exchanges required. No additional exchanges are needed for user tasks or for other RMX/80 extensions.

In the example (see BUFFERS portion), the maximum number of files ever open at once is three, and a static buffer pool is to be supplied by the user. One user buffer is to be located in controller-addressable memory.

Preparing the Configuration Module

General Instructions. An RMX/80 configuration module for a system that includes DFS consists of declarations for the following items:

1.   Task Stacks (RAM)
2.   Task Descriptors (RAM)
3.   Task External Declarations
4.   Initial Task Table (ROM)
5.   Exchange Descriptors (RAM)
6.   Interrupt Exchange Descriptors (RAM)
7.   Initial Exchange Table (ROM)
8.   Create Table (ROM)
9.   Public Data (ROM) if any are required by RMX/80 extension tasks
10. Controller Specification Table (ROM)
11. Device Configuration Table (ROM)
12. Drive Characteristics Table (ROM) if one or more iSBC 204 controllers are included in your system
13. Buffer Allocation Block (ROM) if the DIRSVC task is included in your system

All of the segments listed above are coded from information on Parts I and II of the RMX/80 System Definition Worksheet. Further instructions for coding items 10 through 13 (the items unique to DFS) are given in the sections that follow.

After these instructions, the additional assembly language configuration macros provided for DFS are described. Following the macro descriptions are listings of PL/M and assembly language configuration modules for our example application system.

Controller Specification Table (RQCST). One entry is made in this table for each controller defined on the worksheet. The format of each entry is shown in figure 7-20.
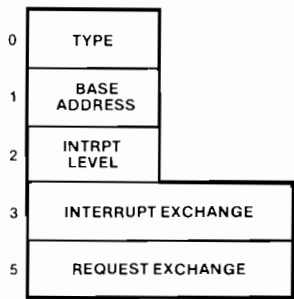

Figure 7-20. Controller Specification Table Entry

Controller TYPE is 0 for iSBC 201, 1 for iSBC 202, or 2 for iSBC 204. The INTER-
RUPT EXCHANGE and REQUEST EXCHANGE fields are the addresses of these
exchanges.

Device Configuration Table (RQDCT). One entry is made in this table for each
drive defined on the worksheet. The format of each entry is shown in figure 7-21.

TYPE refers to the type of controller to which the drive is attached (and also to the
diskette size for iSBC 204 controllers); it is coded as follows:

TYPE	CODE
iSBC 201 and iSBC 204 (standard-size diskette)	0
iSBC 202	1
iSBC 204 (mini-size diskettes)	2

The one-byte variable RQNDEV, which contains the number of entries in the DCT,
is generally defined in this part of the configuration module. (See "PUBLIC Data"
under "Defining System Components" earlier in this chapter.)

Drive Characteristics Table (RQDRC4). This table is required only for systems
that include one or more iSBC 204 controllers. It consists of two parts. The first part
is the first byte in the table and indicates the number of entries in the table. The se-
cond part consists of a series of five-byte entries, one entry for each entry in the
Drive Parameters section of the worksheet — i.e., one entry for each 8271 chip used
in the system.

The format of each table entry is shown in figure 7-22. Note that the last one-byte
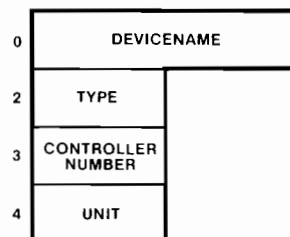field is made up of two four-bit subfields, INDEX COUNT and (HEAD) LOAD
TIME.

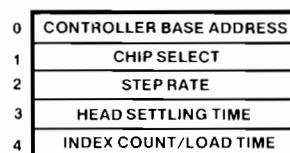Figure 7-21. Device Configuration Table Entry

Figure 7-22. iSBC 204 Drive Characteristics Table Entry

CONTROLLER BASE ADDRESS and CHIP SELECT are as entered on Part II of the System Definition Worksheet. (Refer to "Drive Parameters (iSBC 204 Controller)" under "Defining System Commands" earlier in this chapter.)

The form in which the last three fields (STEP RATE, HEAD SETTLING TIME, and INDEX COUNT/LOAD TIME) must be presented is covered in the iSBC 204 Single Density Diskette Controller Hardware Reference Manual, 9800568. For our example application, note that:

- Time parameters are specified in time units. For standard-size drives a time unit is 1ms; for mini-size drives, 2ms. So for the mini-size drives, STEP RATE, HEAD SETTLING TIME, and LOAD TIME, after being computed in milliseconds, are divided by two and are then rounded up, if necessary, to the next higher integer value.

- Head load time is programmed in intervals of 4 time units; so 35 is programmed as 9 and 40 becomes 10.

- As a consequence of the two situations just discussed, LOAD TIME is divided by 8 for mini-size drives.

Buffer Allocation Block (RQBAB). This segment of the configuration module is optional; if DISKIO is the only DFS service used in the application, the Buffer Allocation Block may be omitted from the configuration module. In all other cases (i.e., the DIRSVC task is included), a Buffer Allocation Block must be coded. The format of the block is shown in figure 7-23.

How to code the Buffer Allocation Block depends on the application's buffer requirements. These fall into three categories, which are discussed in the following paragraphs.

Static Allocation. In this case the user is supplying a buffer pool area, and the block should be coded as follows:

BUFFER SOURCE = zero.

BUFFER RETURN = zero.

NFILES   = maximum number of concurrently open files (from worksheet).

BUFFER POINTER = address of the user-supplied static buffer pool (located in the controller-addressable memory module).
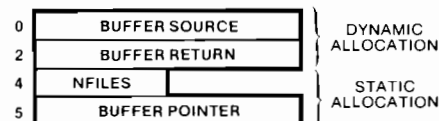

Figure 7-23. Buffer Allocation Block

Dynamic Allocation. In this case DFS is to acquire buffer space at open time from the Free Space Manager. When a file is closed, the space is returned to the Free Space Manager. The Buffer Allocation Block should be coded as follows:

BUFFER SOURCE   = address of the Free Space Manager allocation exchange (RQFSAX).

BUFFER RETURN = address of the Free Space Manager reclamation exchange (RQFSRX).

NFILES   = zero.

BUFFER POINTER = zero.

No Buffers. This case can arise if the application is using a combination of these services only: DELETE, RENAME, ATTRIB, and FORMAT. No additional buffer space is required, and this is communicated to DFS by setting NFILES to zero. (The contents of the other three fields do not matter.) In practice, a system using only these services will rarely occur.

Assembly Language Macros for DFS. In addition to the four configuration macros supplied on the RMX/80 Executive diskettes and described in Chapter 3, RMX/80 provides a set of macros for use in preparing configuration modules for systems including DFS. These macros are on the RMX/80 Extensions diskette in two libraries, RMXEXC.MAC and DFSCFG.MAC. The definitions for these macros can be added to your program module via the assembler $INCLUDE control. When using these macros, specifying a $NOGEN control to suppress the macro expansion printout greatly improves the readability of your listing.

The RMXXCH.MAC library contains the macros XCH, INTXCH, and PUBXCH. The DFSCFG.MAC library contains the macros CONSTD, CST, DCT, DRC4, GENDRC, and BAB. The PUBXCH macros must be coded after the XCH and IN-TXCH macros; and all DCT macros must be coded before the CST macros; and all DRC4 macros must be coded before the GENDRC macros.

Before coding any of these macros, the user must include two SET directives to initialize two counters, NCONT and NDEV, to zero. For systems including one or more iSBC 204 controllers, a third SET directive is required to initialize the counter NCHIPS to zero.

For further information on the parameters required for the macros that generate DFS configuration tables (DCT, CST, and DRC4), refer to "Controller Specification Table", "Device Configuration Table", and "Drive Characteristics Table" earlier in this chapter.

XCH Macro. The XCH macro allocates an exchange with a PUBLIC label, in preparation for building an Initial Exchange Table entry with the PUBXCH macro. The format for a call to XCH is as follows:

    XCH name

Name is the symbolic name assigned to the Exchange Descriptor associated with the Initial Exchange Table entry to be built. The name must conform to the assembly language rules for symbols, and its second character should not be "Q" or "?".

INTXCH Macro. The INTXCH macro allocates an interrupt exchange with a PUBLIC label, in preparation for building an Initial Exchange Table entry with the PUBXCH macro. The format for a call to INTXCH is as follows:

    INTXCH name

Name is the symbolic name assigned to the Interrupt Exchange Descriptor associated with the Initial Exchange Table entry to be built. The name must conform to the assembly language rules for symbols, and its second character should not be "Q" or "?".

PUBXCH Macro. The PUBXCH macro builds an Initial Exchange Table entry. It is identical to the XCHADR macro described in Chapter 3, except that it is for use with the XCH and INTXCH macros and does not declare the exchange to be external. The format for a call to PUBXCH is as follows:

    PUBXCH name

Name is the symbolic name assigned to the Exchange Descriptor or Interrupt Exchange Descriptor associated with the Initial Exchange Table entry to be built. The name must conform to the assembly language rules for symbols, and its second character should not be "Q" or "?".

CONSTD Macro. The CONSTD macro builds a Static Task Descriptor for a controller task. It is similar to the STD macro described in Chapter 3, with one major difference: it does not generate a stack for the controller task, since the controller task stack must be in the controller-addressable memory module. An external reference is generated for the stack address. Another difference from STD is that the optional *tdxtra* field is not included. The format for a call to CONSTD is as follows:

    CONSTD name,entry,stklen,stkadr,pri,exch

Name is the symbolic name assigned to the procedure associated with this Static Task Descriptor, and is the entry address for the task. The name must conform to the assembly language rules for symbols, and its second character should not be "Q" or "?".

Entry is the entry address of the controller task.

Stklen is the length (in bytes) of the external stack for the task.

Stkadr is the address of the external stack.

Pri is the priority of the task.

Exch is the request exchange (default exchange) to be associated with this task. This is a required parameter for controller tasks and must be the same exchange defined for the corresponding controller(s) in the Controller Specification Table (i.e., in the *reqxch* parameter of the CST macro(s)) for those controller(s).

DCT Macro. The DCT macro builds one entry in the Device Configuration Table (RQDCT). The format for a call to DCT is as follows:

    DCT dvname,dvtype,cont,unit

Dvname is the two-character device name (without quotes).

Dvtype is the code for the device type (see "Device Configuration Table" earlier in this chapter).

Cont is the controller number as entered on the System Definition Worksheet.

Unit is the unit number as entered on the worksheet.

CST Macro. The CST macro builds one entry in the Controller Specification Table (RQCST). The format for a call to CST is as follows:

    CST ctype,ioadr,intlvl,intxch,reqxch

Ctype is the controller type (see "Controller Specification Table" earlier in this chapter).

Ioadr is the controller base address.

Intlvl is the interrupt level to be associated with this controller.

Intxch is the address (RQLnEX) of the interrupt exchange associated with *intlvl*.

Reqxch is the address of the request exchange (default exchange) to be associated with the controller. This is a required parameter for CST entries, and must be the same exchange defined for the corresponding controller task (in the *exch* parameter of the CONSTD macro that builds that task).

DRC4 Macro. The DRC4 macro specifies one entry in the Drive Characteristics Table for systems using iSBC 204 controllers. (Each table entry corresponds to an 8271 controller chip on the iSBC 204.) The format for a call to DRC4 is as follows:

    DRC4 base,chip,steprt,headst,idxldt

Base is the controller base address.

Chip is the chip select (0 or 1).

Steprt is the controller chip step rate parameter.

Headst is the controller chip head settling time parameter.

Idxldt is the controller chip index count (high-order four bits) and head load time (low-order four bits) parameter.

GENDRC Macro. The GENDRC macro builds the Drive Characteristics Table for systems using iSBC 204 controllers. The format for a call to GENDRC is as follows:

    GENDRC

GENDRC refers to data defined by the DRC4macros; therefore, the call to GEN-DRC must follow these macros.

Operands are not permitted with the GENDRC macro.

BAB Macro. The BAB macro builds the Buffer Allocation Block (RQBAB). The format for a call to BAB is as follows:

    BAB nfiles [,poolad]

Nfiles specifies the maximum number of concurrently open files (from the System Definition Worksheet) if static buffer allocation is selected. For dynamic buffer allocation (using the Free Space Manager), or for the "no buffers" case (DELETE, RENAME, ATTRIB, and/or FORMAT services only), *nfiles* must be zero.

Poolad is the RAM address of the user-supplied static buffer pool (located in the controller-addressable memory module).

Note that although the Buffer Allocation Block has four fields, only two parameters—only one parameter for dynamic buffer allocation or the "no buffers" case—are needed to specify the content of these fields. For dynamic buffer allocation or no buffers (*nfiles* =0), the BAB macro supplies the addresses of the Free Space Manager allocation and reclamation exchanges in the appropriate fields.

PL/M Configuration Module for Example Application.

```
DFS$CONFIG: DO;

/* FIRST DECLARE SOME SYMBOLIC VALUES TO MAKE LATER
 * CODING EASIER. NOTE THAT MOST OF THESE ARE
 * DISTRIBUTED ON THE SYSTEM DISK AND CAN BE
 * $INCLUDED IN THE USER'S PROGRAM (SEE APPENDIX D).
 */

DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        MESSAGE$HEAD          ADDRESS,
        MESSAGE$TAIL          ADDRESS,
        TASK$HEAD             ADDRESS,
        TASK$TAIL             ADDRESS,
        EXCHANGE$LINK         ADDRESS)';

DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        MESSAGE$HEAD          ADDRESS,
        MESSAGE$TAIL          ADDRESS,
        TASK$HEAD             ADDRESS,
        TASK$TAIL             ADDRESS,
        EXCHANGE$LINK         ADDRESS,
        LINK                  ADDRESS,
        LENGTH                ADDRESS,
        TYPE                  BYTE)';

DECLARE TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
        DELAY$LINK$FORWARD    ADDRESS,
        DELAY$LINK$BACK       ADDRESS,
        THREAD                ADDRESS,
        DELAY                 ADDRESS,
        EXCHANGE$ADDRESS      ADDRESS,
        SP                    ADDRESS,
        MARKER                ADDRESS,
        PRIORITY              BYTE,
        STATUS                BYTE,
        NAME$PTR              ADDRESS,
        TASK$LINK             ADDRESS)';
```

```
                    DECLARE STATIC$TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
                            NAME(6)                     BYTE,
                            PC                          ADDRESS,
                            SP                          ADDRESS,
                            STKLEN                      ADDRESS,
                            PRIORITY                    BYTE,
                            EXCHANGE$ADDRESS            ADDRESS,
                            TASK$PTR                    ADDRESS)';

                    DECLARE CREATE$TABLE                LITERALLY 'STRUCTURE (
                            TASK$POINTER                ADDRESS,
                            TASK$COUNT                  BYTE,
                            EXCHANGE$POINTER            ADDRESS,
                            EXCHANGE$COUNT              BYTE)';

                    DECLARE /* CONTROLLER SPECIFICATION TABLE ENTRY */
                            CST$ENTRY                   LITERALLY 'STRUCTURE (
                            CONTROLLER$TYPE             BYTE,
                            IO$BASE$ADDR                BYTE,
                            INTRP$LEVEL                 BYTE,
                            INTRP$XCH                   ADDRESS,
                            REQUEST$XCH                 ADDRESS)';

                    DECLARE /* DEVICE CONFIGURATION TABLE ENTRY */
                            DCT$ENTRY                   LITERALLY 'STRUCTURE (
                            DEVICE$NAME(2)              BYTE,
                            DEVICETYPE                  BYTE,
                            CONTROLLER                  BYTE,
                            UNIT                        BYTE)';

                    /****  STACK LENGTHS  ****/
                    DECLARE DSK$STL LITERALLY '48';          /* DISKIO */
                    DECLARE DIR$STL  LITERALLY '48';          /* DIRSVC */
                    DECLARE U1$STL   LITERALLY '64';          /* UTASK1 */
                    DECLARE U2$STL   LITERALLY '80';          /* UTASK2 */
                    DECLARE CN$STL   LITERALLY '80';          /* CNTRLRS */


                    /****  TASK PRIORITIES  ****/
                    DECLARE DSK$PRI LITERALLY '129';          /* DISKIO */
                    DECLARE DIR$PRI  LITERALLY '135';          /* DIRSVC */
                    DECLARE CN1$PRI LITERALLY ' 33';          /* CNTRL1 */
                    DECLARE CN2$PRI LITERALLY ' 49';          /* CNTRL2 */
                    DECLARE CN3$PRI LITERALLY ' 66';          /* CNTRL3 */
                    DECLARE U1$PRI   LITERALLY '150';          /* UTASK1 */
                    DECLARE U2$PRI   LITERALLY '160';          /* UTASK2 */

                    /****  TASK STACKS  ****/
                    DECLARE DSK$STK(DSK$STL) BYTE;           /* DISKIO */
                    DECLARE DIR$STK(DIR$STL)  BYTE;           /* DIRSVC */
                    DECLARE U1$STK(U1$STL)    BYTE;           /* UTASK1 */
                    DECLARE U2$STK(U2$STL)    BYTE;           /* UTASK2 */

                    /****  CONTROLLER STACKS ARE IN CAMMOD  ****/
                    DECLARE  CN1$STK(1)        BYTE EXTERNAL;
                    DECLARE  CN2$STK(1)        BYTE EXTERNAL;
                    DECLARE  CN3$STK(1)        BYTE EXTERNAL;
```

```
/***  TASK DESCRIPTORS  ****/
DECLARE DSK$TD TASK$DESCRIPTOR; /* DISKIO */
DECLARE DIR$TD  TASK$DESCRIPTOR; /* DIRSVC */
DECLARE CN1$TD TASK$DESCRIPTOR; /* CNTRL1 */
DECLARE CN2$TD TASK$DESCRIPTOR; /* CNTRL2 */
DECLARE CN3$TD TASK$DESCRIPTOR; /* CNTRL4 */
DECLARE U1$TD   TASK$DESCRIPTOR; /* UTASK1 */
DECLARE U2$TD   TASK$DESCRIPTOR; /* UTASK2 */

/****  TASK EXTERNAL DECLARATIONS  ****/
RQPDSK:       PROCEDURE EXTERNAL;/* DISKIO */
END RQPDSK;
RQPDIR:       PROCEDURE EXTERNAL;/* DIRSVC */
END RQPDIR;
RQHD1:        PROCEDURE EXTERNAL;/* CNTRL1 */
END RQHD1;
RQHD4:        PROCEDURE EXTERNAL;/* CNTRL2 & CNTRL3 */
END RQHD4;
UTASK1:       PROCEDURE EXTERNAL;/* UTASK1 */
END UTASK1;
UTASK2:       PROCEDURE EXTERNAL;/* UTASK2 */
END UTASK2;


/****  INITIAL TASK TABLE  ****/
DECLARE   NTASK         LITERALLY '7';
DECLARE   ITT(NTASK)    STATIC$TASK$DESCRIPTOR  DATA (

/* STD FOR DISKIO */
'DISKIO',              /*TASK NAME */
.RQPDSK,               /* INITIAL P.C. (ENTRY POINT)*/
.DSK$STK,              /* STACK POINTER */
DSK$STL,               /* STACK LENGTH */
DSK$PRI,               /* PRIORITY */
.RQDSKX,               /* DEFAULT EXCHANGE */
.DSK$TD,               /* TASK DESCRIPTOR */

/* STD FOR DIRECTORY SERVICES */
'DIRSVC',              /*TASK NAME */
.RQPDIR,               /* INITIAL P.C. */
.DIR$STK,              /* STACK POINTER */
DIR$STL,               /* STACK LENGTH */
DIR$PRI,               /* PRIORITY */
.RQDIRX,               /* DEFAULT EXCHANGE */
.DIR$TD,               /* TASK DESCRIPTOR */

/* STD FOR CONTROLLER TASK 1 */
'CNTRL1',              /*TASK NAME */
.RQHD1,                /* INITIAL P.C. */
.CN1$STK,              /* STACK POINTER */
CN$STL,                /* STACK LENGTH */
CN1$PRI,               /* PRIORITY */
.CNTL1X,               /* DEFAULT EXCHANGE */
.CN1$TD,               /* TASK DESCRIPTOR */
```

```
/* STD FOR CONTROLLER TASK 2 */
'CNTRL2',              /*TASK NAME */
.RQHD4,                /* INITIAL P.C. */
.CN2$STK,              /* STACK POINTER */
CN$STL,                /* STACK LENGTH */
CN2$PRI,               /* PRIORITY */
.CNTL2X,               /* DEFAULT EXCHANGE */
.CN2$TD,               /* TASK DESCRIPTOR */

/* STD FOR CONTROLLER TASK 3 */
'CNTRL3',              /*TASK NAME */
.RQHD4,                /* INITIAL P.C. */
.CN3$STK,              /* STACK POINTER */
CN$STL,                /* STACK LENGTH */
CN3$PRI,               /* PRIORITY */
.CNTL3X,               /* DEFAULT EXCHANGE */
.CN3$TD,               /* TASK DESCRIPTOR */

/* STD FOR USER TASK 1 */
'UTASK1',              /*TASK NAME */
.UTASK1                /* INITIAL P.C. */
.U1$STK,               /* STACK POINTER */
U1$STL,                /* STACK LENGTH */
U1$PRI,                /* PRIORITY */
.UTSK1X,               /* DEFAULT EXCHANGE */
.U1$TD,                /* TASK DESCRIPTOR */

/* STD FOR USER TASK 2 */
'UTASK2',              /*TASK NAME */
.UTASK2,               /* INITIAL P.C. */
.U2$STK,               /* STACK POINTER */
U2$STL,                /* STACK LENGTH */
U2$PRI,                /* PRIORITY */
.UTSK2X,               /* DEFAULT EXCHANGE */
.U2$TD);               /* TASK DESCRIPTOR */

/**** EXCHANGE DESCRIPTORS ****/
DECLARE RQDSKX EXCHANGE$DESCRIPTOR EXTERNAL; /* DISKIO */
DECLARE RQDIRX  EXCHANGE$DESCRIPTOR :          /* DIRSVC */
DECLARE CNTL1X  EXCHANGE$DESCRIPTOR :          /* CNTRL1 */
DECLARE CNTL2X  EXCHANGE$DESCRIPTOR :          /* CNTRL2 */
DECLARE CNTL3X  EXCHANGE$DESCRIPTOR :          /* CNTRL3 */
DECLARE UTSK1X  EXCHANGE$DESCRIPTOR :          /* UTASK1 */
DECLARE UTSK2X  EXCHANGE$DESCRIPTOR :          /* UTASK2 */

/**** INTERRUPT EXCHANGE DESCRIPTORS ****/
DECLARE RQL2EX  INT$EXCHANGE$DESCRIPTOR PUBLIC; /* LEVEL #2 */
DECLARE RQL3EX  INT$EXCHANGE$DESCRIPTOR PUBLIC; /* LEVEL #3 */
DECLARE RQL4EX  INT$EXCHANGE$DESCRIPTOR PUBLIC; /* LEVEL #4 */
```

```
/****   INITIAL  EXCHANGE TABLE  ****/
DECLARE  NEXCH   LITERALLY '10',
DECLARE  IET(NEXCH)          ADDRESS DATA (
          .RQDSKX,           /* DISKIO */
          .RQDIRX,           /* DIRSVC */
          .CNTL1X,           /* CNTRL1 */
          .CNTL2X,           /* CNTRL2 */
          .CNTL3X.           /* CNTRL3 */
          .UTSK1X,           /* UTASK1 */
          .UTSK2X,           /* UTASK2 */
          .RQL2EX,           /* LEVEL 2 INTERRUPT */
          .RQL3EX,           /* LEVEL 3 INTERRUPT */
          .RQL4EX);          /* LEVEL 4 INTERRUPT */

/****   CREATE TABLE  ****/
DECLARE  RQCRTB CREATE$TABLE PUBLIC DATA (
          .ITT
          NTASK,
          .IET,
          NEXCH);

/****   CONTROLLER SPECIFICATION TABLE  ****/
DECLARE NCNTRL          LITERALLY '3';
DECLARE RQCST(NCNTRL)   CST$ENTRY PUBLIC DATA (
/*    TYPE      BASE      INTRPT      INTRPT        REQUEST      */
/*              ADDR      LEVEL       EXCH          EXCH         */
       0,       88H,        2,       .RQL2EX,      .CNTL1X,
       1,       70H,        3,       .RQL3EX,      .CNTL2X,
       2,       60H,        4,       .RQL4EX,      .CNTL3X;

/****   DEVICE CONFIGURATION TABLE  ****/
DECLARE  NDEV          LITERALLY '5';
DECLARE  RQNDEV        BYTE PUBLIC DATA (NDEV);
DECLARE  RQDCT(NDEV) DCT$ENTRY PUBLIC DATA (
          /* DEVICENAME          TYPE       CONTROLLER         UNIT */
              'SA',               0,            0,              0 ,
              'SB',               0,            0,              1 ,
              'SC',               0,            1,              0 ,
              'M1',               2,            2,              0 ,
              'M2',               2,            2,              1);

/****   DRIVE CHARACTERISTICS TABLE FOR ISBC 204  ****/
DECLARE  RQDRC4(11) BYTE PUBLIC DATA (
     /* NUMBER OF ENTRIES */
                        2,
/*    CTLR BASE      CHIP SEL       STEP RATE       HD SET       IDX/LOAD    */
       70H,            0,             8,              10,          A9H,
       60H,            0,            20,              10,          AAH);

/****   REFERENCE TO BUFFER POOL DEFINED IN CAMMOD  ****/
DECLARE      BUF$POOL(1)        BYTE EXTERNAL;

/****   BUFFER ALLOCATION BLOCK  ****/
DECLARE      RQBAB                STRUCTURE (
             BUF$SOURCE           ADDRESS,
             BUF$RETURN           ADDRESS,
             NFILES               BYTE,
             BUF$POINTER          ADDRESS) PUBLIC DATA (0,0,3,.BUF$POOL);

     END DFS$CONFIG;
```

Assembly-Language Configuration Module For Example Application

```
                NAME    DFSCFG
                CSEG
; INCLUDE MACRO DEFINITIONS FROM PRODUCT DISKETTES
$INCLUDE (:F2:STD.MAC)
$INCLUDE (:F2:XCHADR.MAC)
$INCLUDE (:F2:CRTAB.MAC)
$INCLUDE (:F2:GENTD.MAC)
$INCLUDE (:F2:RMXXCH.MAC)
$INCLUDE (:F2:DFSCFG.MAC)
NTASK   SET     0
NEXCH   SET     0
NDEV    SET     0
NCONT   SET     0
NCHIPS  SET     0
;
; BUILD THE INITIAL TASK TABLE (ITT)
;
                STD     RQPDSK,48,129,RQDSKX
                STD     RQPDIR,48,135,RQDIRX
                EXTRN   RQHD1
                EXTRN   RQHD4
                CONSTD  CONTR1,RQHD1,80,CN1STK,33,CNTL1X
                CONSTD  CONTR2,RQHD4,80,CN2STK,49,CNTL2X
                CONSTD  CONTR3,RQHD4,80,CN3STK,66,CNTL3X
                STD     UTASK1,64,150,UTSK1X
                STD     UTASK2,80,160,UTSK2X
;
; ALLOCATE THE TASK DESCRIPTORS
;
                GENTD
;
; ALLOCATE ALL EXCHANGES
;
                XCH     CNTL1X
                XCH     CNTL2X
               ·XCH     CNTL3X
                INTXCH  RQL2EX
                INTXCH  RQL3EX
                INTXCH  RQL4EX
;
; BUILD THE INITIAL EXCHANGE TABLE (IET)
;
                XCHADR  RQDSKX
                XCHADR  RQDIRX
                PUBXCH  CNTL1X
                PUBXCH  CNTL2X
                PUBXCH  CNTL3X
                XCHADR  UTSK1X
                XCHADR  UTSK2X
                PUBXCH  RQL2EX
                PUBXCH  RQL3EX
                PUBXCH  RQL4EX
;
; BUILD CREATE TABLE
;
                CRTAB
;
```

```
; BUILD DEVICE CONFIGURATION TABLE
;
        DCT     SA,0,0,0
        DCT     SB,0,0,1
        DCT     SC,0,1,0
        DCT     M1,2,2,0 ·
        DCT     M2,2,2,1
;
; BUILD CONTROLLER SPECIFICATION TABLE
;
        CST     0,88H,2,RQL2EX,CNTL1X
        CST     1,70H,3,RQL3EX,CNTL2X
        CST     2,60H,4,RQL3EX,CNTL3X
;
; BUILD DRIVE CHARACTERISTICS TABLE FOR iSBC 204
;
        DRC4    70H,0,8,10,A9H
        DRC4    70H,0,20,10,AAH
;
; ALLOCATE THE DRIVE CHARACTERISTICS TABLE
;
        GENDRC
;
; SET UP BUFFER ALLOCATION BLOCK
;
        BAB     3,BUFPOL
        END
```

Preparing the Controller-Addressable Memory Module

For iSBC 80/10 and 80/20 systems, all memory locations which must be accessible to disk drive controllers are coded in the controller-addressable memory module. This includes:

- controller task stacks
- DFS internal buffer space
- DFS static buffer pool (optional)
- controller-addressable user buffers (optional)

Nothing else should be coded in the module.

After the module has been compiled or assembled, it is located at the highest possible controller-addressable RAM address (off-board for iSBC 80/20 and 80/10 systems). When it is subsequently linked and located with the rest of the system modules, its location is not changed. All data defined in the module must be PUBLIC. No particular module name is required.

Controller Task Stacks. One 80-byte stack must be defined for each controller task. The format is the same as the stacks in the configuration module. DFS uses the stacks to pass parameter blocks to the controllers.

DFS Internal Buffer Space (RQDBUF). A 700-byte area called RQDBUF must be supplied for DFS if the DIRSVC task is included in the application.

DFS Static Buffer Pool. If static buffer allocation has been selected, a buffer pool area must be provided. Appendix E shows how much space is required to open a file. The size of the buffer pool area must be this amount times the number of files entered on the worksheet.

Controller-Addressable User Buffers. Any user buffers which can be accessed by disk controllers must be located in this module.

Example. The following shows how a controller-addressable memory module is coded in PL/M for the sample application:

```
CAMMOD: DO;
/* CONTROLLER TASK STACKS */
DECLARE       CN1$STK (80)         BYTE PUBLIC;
DECLARE       CN2$STK (80)         BYTE PUBLIC;
DECLARE       CN3$STK (80)         BYTE PUBLIC;

/* DFS INTERNAL BUFFER SPACE */
DECLARE       RQDBUF(700)          BYTE PUBLIC;

/* DFS STATIC BUFFER POOL */
DECLARE       BUF$POOL(1200)       BYTE PUBLIC;

/* CONTROLLER-ADDRESSABLE USER BUFFERS */
DECLARE       UBUF2(256)           BYTE PUBLIC;

END CAMMOD;
```

Locating the Controller-Addressable Memory Module. Prior to linking the system, the controller-addressable memory module should be located at the highest possible RAM address. To determine what this address is, subtract the module length from the highest RAM location in the system. The length of the module can be determined by three methods:

*   Add up the lengths of all the variables as defined in the source code.

*   The PL/M compiler outputs the length, calling it VARIABLE AREA SIZE.

*   The module can be located without specifying segment addresses; if this is done, the length of the module will be listed in the memory map as the length of the DATA segment.

After the module's base address has been determined by subtracting the module length from the system's highest RAM location, the controller-addressable memory module should be located as follows:

LOCATE :Fn: CAM object file DATA(base address) STACKSIZE(0)

For example, assume that the highest RAM address in the sample application system is 7FFFH and that the length of the controller-addressable memory module is 952H. The module is located as follows:

LOCATE :F1:CAMMOD.OBJ DATA(76ADH) STACKSIZE(0)

In this example the name of the located module will be :F1:CAMMOD.

Linking and Locating

General Instructions for Linking. Here is the general form of the LINK command for an RMX/80 iSBC 80/20 system which includes the Disk File System (sequence is significant):

```
LINK      :Fn:RMX820.LIB(START), &
          :Fn: configuration module, &
          :Fn: user task modules, &
          :Fn:DFSDIR.LIB(modules), &
          :Fn:DIO820.LIB, &
          :Fn:DFSUNR.LIB, &
          :Fn: controller-addressable memory module, &
          :Fn: other RMX/80 extension tasks, &
          :Fn:RMX820.LIB, &
          :Fn:UNRSLV.LIB, &
          :Fn:PLM80.LIB TO :Fn: load module
```

For iSBC 80/10-based or iSBC 80/30-based systems, substitute "810.LIB" or "830.LIB" for "820.LIB" where it appears in the LINK list above.

The following paragraphs explain how to code the DFS libraries in the LINK command.

DFSDIR.LIB. This library is optional and may be omitted if DISKIO is the *only* DFS service used in the application. If *all* DFS services are used in the application, then module names need not be specified for the library — that is, the statement can be coded as:

```
DFSDIR.LIB, &
```

In all other cases, the modules to be included in the system must be specified. Module names correspond to DFS service task names as follows:

TASK	MODULE
—	SEEK
DIRSVC	DIRECTORY
ATTRIB	ATTRIB
DELETE	DELETE
RENAME	RENAME
LOAD	LOAD
FORMAT	FORMAT
	FORMAT201
	FORMAT202
	FORMAT204
	FMTTABLE

Recall that SEEK is not coded in the configuration module as a task; it is linked into the system load module from the DFSDIR.LIB library. In order to use the FORMAT service, the FORMAT module must be specified. In addition, FORMAT201, FORMAT202, and/or FORMAT204 must be linked in, depending on which types(s) of devices — standard-size single-density, standard-size double-density, or mini-size — are to be formatted. (Note that if you have an iSBC 204 controller, you use FORMAT201 for standard-size drives, or FORMAT204 for mini-size drives.)

If you do not specify DFS services individually, all modules in the list above— including FORMAT201, FORMAT202, *and* FORMAT204—will be linked in automatically. You will probably rarely need all of these modules in your system. For this reason, it is generally advisable to specify the services you need, in order to minimize memory requirement for your system.

DIO820.LIB, DIO830.LIB, or DIO810.LIB. One or the other of these libraries must be specified depending on which computer—iSBC 80/20, 80/30, or 80/10—is being used. This library contains the DISKIO modules and the controller handler modules.

DFSUNR.LIB. This library must always be specified. It contains declarations to take care of the unresolved references which occur when not all DFS services are linked into the system.

Example. Here is how the example application is linked assuming the computer is an iSBC 80/20, all files reside on device F1, the configuration module is called DFSCON.OBJ, and the load module is called DFSSYS.LNK:

```
LINK    :F1:RMX820.LIB(START), &
        :F1:DFSCON.OBJ, &
        :F1:UTASK1.OBJ, &
        :F1:UTASK2.OBJ, &
        :F1:DFSDIR.LIB(DIRECTORY,SEEK), &
        :F1:DIO820.LIB, &
        :F1:DFSUNR.LIB, &
        :F1:CAMMOD, &
        :F1:RMX820.LIB, &
        :F1:UNRSLV.LIB, &
        :F1:PLM80.LIB TO :F1:DFSSYS.LNK
```

Locating. No special information is needed to locate object code for a system that includes DFS; for general instructions, refer to Chapter 3.

The following command will locate our example application system.

```
LOCATE :F1:DFSSYS.LNK TO :F1: DFSSYS.LOC &
CODE(0) DATA (3800H) STACKSIZE(0) START(0)
```

General Description

The Analog Handlers provide analog input and output services for users whose systems interface with the iSBC 711, iSBC 724, and iSBC 732 analog boards. The Input Handler performs repetitive, sequential, and random channel input; the Output Handler performs random channel output.

Use of the Analog Handlers requires a basic degree of familiarity with the analog board(s) being used. This section is written assuming such familiarity. For details on the analog boards, refer to the appropriate Hardware Reference Manuals:

- iSBC 711 Analog Input Board Hardware Reference Manual, 9800485A

- iSBC 724 Analog Output Board Hardware Reference Manual, 9800486A

- iSBC 732 Combination Analog Input/Output Board Hardware Reference Manual, 9800487A

Functions Provided

Repetitive Channel Input. This function repetitively samples a single A/D channel up to 254 times, applying the same gain each time. The data values obtained are placed in a user-specified array.

Sequential Channel Input with Single Gain. This function samples a sequence of consecutively numbered A/D channels, applying the same gain to every sample. The data values obtained are placed in a user-specified array.

Sequential Channel Input with Variable Gain. This function samples a sequence of consecutively numbered A/D channels, applying an individually specified gain to each sample. The data values obtained are placed in a user-specified array.

Random Channel Input. This function samples a set of A/D channels whose order is user-specified, applying an individually specified gain to each sample. The data values obtained are placed in a user-specified array.

Random Channel Output. This function simply sets a number of D/A channels. Both the channels and their outputs are user-specified.

Use Environment

The Analog Handlers run under RMX/80 in an iSBC 80/20, 80/30, or 80/10 environment. The Handlers interface with an iSBC 711, iSBC 724, or iSBC 732, or a combination of these boards.

A single version of the Analog Handlers supports the iSBC 80/20, 80/30, and 80/10.

Memory and Hardware Requirements

The Analog Input Handler requires less than 600 bytes of code space and less than 30 bytes of data space, and uses a 34-byte stack. The Analog Output Handler requires less than 200 bytes of code space and less than 20 bytes of data space, and uses a 32-byte stack. Refer to Appendix D for exact byte counts.

Analog boards are factory configured with certain features fixed, such as the address of the base register on the A/D or D/A converter. If more than one analog board is installed in his system, the user must reconfigure the base register address of each additional board as described in the Hardware Reference Manual for that board. (Refer to the list of manuals at the beginning of this chapter.)

How the Analog Handlers Operate

The Analog Handlers extension to RMX/80 consists of two tasks, an Input Handler and an Output Handler. These tasks are independent of each other, so that if only analog input or only analog output is needed, only one of the tasks need be configured in the system.

As shown in figure 8-1, each task waits at an exchange — RQAIEX for the Input Handler, RQAOEX for the Output Handler — for a request message from the user task. When an Analog Hander task receives a message, it performs the appropriate operation, modifies the request message, and returns the message to the appropriate user-defined response exchange, designated in figure 8-1 as RESP$IEX or RESP$OEX.
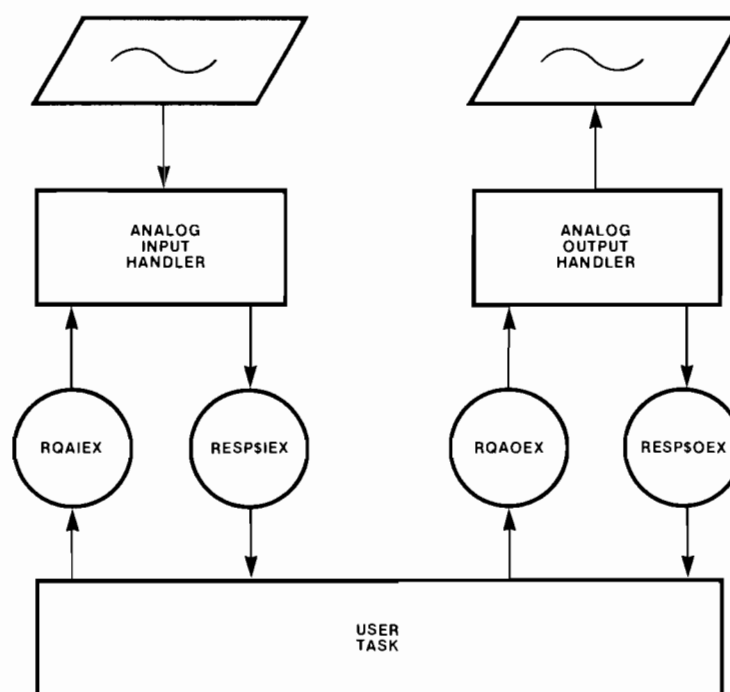
Figure 8-1. Exchanges for Analog Handlers

Using the Analog Handlers

General Usage Considerations

The sequence of events that occurs when a user task sends a message to the exchange of an Analog Handler depends on the relative priorities of the user task and the Handler. The software priority of each Handler may be specified by the user. In most applications it probably should be higher than the user task. This means that when the user task sends a request message to an Analog Handler exchange, the Handler will begin running immediately. However, if the user task is assigned a

higher priority, sending a message will make the Handler ready, and it will begin to run when it becomes the highest-priority ready task.

While each operation is in progress, the Analog Handler has control of the processor. This means that other processing cannot be accomplished concurrently with the analog input or output operation. However, the Analog Handlers do not disable interrupts, and servicing of a higher-priority interrupt may increase the time interval between two input samplings or the presentation of two analog outputs. If this is not acceptable, the tasks should run at a software priority in the range 1-16 (1-4 on the iSBC 80/30 and 1-128 on the iSBC 80/10) in order to lock out all interrupts. Note that in an iSBC 80/10 system with an external clock running, this may result in missing one or more clock ticks if the time base (time between clock interrupts) is less than the time required to perform the input or output.

Error Conditions

The Analog Handler tasks do not check the validity of input parameters, and the operation resulting from invalid arguments is undefined. Such error checking has been omitted in order to minimize response time.

The only hard error detected is failure of the A/D converter to respond. If this occurs, the Analog Input Handler sets the STATUS field in the request message to 3 before it returns the message to the response exchange.

Performance Data

Use of the Analog Handlers with RMX/80 considerably reduces the programming time and effort required for your system. However, the Analog Handlers do introduce some system overhead, and thus may not be an acceptable solution for systems in which speed is extremely critical.

Table 8-1 provides data on the time required to process an analog input or output request of each type. The setup time is incurred once for the request; the time per iteration is incurred once each time a channel is sampled. Thus, for example, a request for repetitive sampling of a particular channel 10 times would take 2.28 + 10(0.22) msec, or 4.48 msec. These performance times apply when the code is in on-board RAM or ROM, and the data is in off-board RAM. Setup times include both the RQSEND and the RQWAIT operations as well as the analog processing.

Table 8-1. Performance Times for Analog Handler Functions

Function	Setup Time (msec)	Time per Iteration (msec)
Repetitive Channel Input	2.28	0.22
Sequential Channel Input with Single Gain	2.28	0.22
Sequential Channel Input with Variable Gain	2.41	0.29
Random Channel Input	2.41	0.29
Random Channel Output	1.9	0.3

Requests for Analog Input and Output

To request analog input or output processing, the user task must create and send a message to the appropriate Analog Handler exchange — RQAIEX for input, or

RQAOEX for output. The request message includes pointers to other data structures, which must be built before the message is sent. This section describes the request message and other data structures associated with each function, and also gives an example showing how to code a request for that function.

The coding examples are intended to be succinct illustrations of the essential use of each input or output function; they should not be construed as complete, executable user tasks. In order to make the usage of the services as clear as possible, the examples have been coded in PL/M. (These examples imply use of the INCLUDE file AITYP.ELT or AOTYP.ELT for symbolic values of message types; refer to Appendix A.) The first example (repetitive channel input), has also been coded in assembly language to show the corresponding constructs. There is a high degree of commonality in the user interfaces to the various Analog Handler functions; it should be easy for assembly language programmers to "translate" the other examples as necessary.

Each example includes an error routine that is empty except for a comment that states that user code should replace the comment. This approach has been taken in order to emphasize that user tasks must be prepared for errors, but that the technique used for error processing is necessarily application-dependent. Error routines should, at least, check for a STATUS value of 3, indicating failure of the converter to respond. The user task may check for input values that might be invalid and/or for an ACTUAL value that differs from COUNT.

Repetitive Channel Input.

Request Message Format. A task requesting that a single A/D channel be sampled repetitively creates and sends (to the RQAIEX exchange) a message with the format shown in figure 8-2. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by the Analog Input Handler when it returns the message.

| 0	LINK	
2	LENGTH*	
4	TYPE*	
5	HOME EXCHANGE	
7	RESPONSE EXCHANGE*	
9	STATUS	
11	BASE REG POINTER*	
13	CHAN-GAIN*	not used
15	DATA ARRAY POINTER*	
17	COUNT*	
19	ACTUAL	
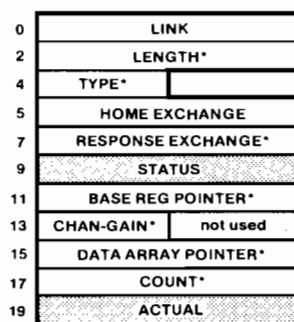
Figure 8-2. Request Message for Repetitive Channel Input

LENGTH is 21.

TYPE is AIREP (30).

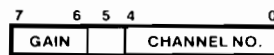HOME EXCHANGE is not used by the Analog Input Handler.

RESPONSE EXCHANGE must specify the address of the user-defined exchange to which the Analog Input Handler is to send the message when the operation is completed.

When the Analog Input Handler returns this message, STATUS is set to 1, 2, 3, or 4, to indicate one of the following conditions:

1:   all data collected.
2:   (reserved for future use.)
3:   operation aborted; converter did not respond.
4:   (reserved for future use.)

BASE REG POINTER must specify the address of the Command-Status Register of the selected A/D converter.

CHAN-GAIN must specify the number of the channel to be sampled and the gain that is to be applied to each sample. Its format is:

```
 7    6  5  4                  0
┌──────┬─┬──────────────────────┐
│ GAIN │ │   CHANNEL NO.        │
└──────┴─┴──────────────────────┘
```

CHANNEL NO., bits 0-4: a number 0-31.
GAIN, bits 6-7: a value of 0, 1, 2, or 3; represents a gain of x1, x2, x4, or x8, respectively.
Bit 5 is ignored.

DATA ARRAY POINTER must specify the address of the array to receive the sampled inputs. The number of elements in the array is equal to COUNT, and each element is two bytes. The Analog Input Handler will store the ith sample in the ith element of the array (recall that the ith element of the array has array index i-1) , in the following format:

```
15                             4 3        0
┌────────────────────────────────┬─────────┐
│            DATA                 │ 0 0 0 0 │
└────────────────────────────────┴─────────┘
```

Each element represents the actual value read from the A/D converter Data Register — 12 bits of data, followed by 4 zeros in the least significant bit positions. The voltage level corresponding to the 12-bit number is dependent upon the full-scale voltage range and the binary code, both of which are jumper-selected by the user as described in the Hardware Reference Manual for the analog input board. (See list of manuals at the beginning of this chapter.)

COUNT must specify the number of samples to be collected. Permissible values are 1-254.

When the Analog Input Handler returns this message, ACTUAL is set to the actual number of samples collected.

Example. The following coding example takes five samplings from a single analog channel (channel 31) and stores the result in the array CURR$TEMP (CTEMP in assembly language). The gain applied is x2. This code could be used to update the roller temperature reading in the paper factory process control example discussed in Chapter 1.

```
/* EXAMPLE OF REPETITIVE CHANNEL INPUT (PL/M) */

/* REQUEST MESSAGE */
DECLARE REP$INP$MSG STRUCTURE
    (LINK                       ADDRESS,
    LENGTH                      ADDRESS,
    TYPE                        BYTE,
    HOME$EXCHANGE               ADDRESS,
    RESPONSE$EXCHANGE           ADDRESS,
    STATUS                      ADDRESS,
    BASE$REG$POINTER            ADDRESS,
    CHAN$GAIN                   BYTE,
    UNUSED                      BYTE,
    DATA$ARRAY$POINTER          ADDRESS,
    COUNT                       ADDRESS,
    ACTUAL                      ADDRESS);
    /* ASSUMED CONTENT OF MESSAGES */
    /* 0, 21, AIREP, 0,.RESP$EX, 0, 0F700H, 01011111B,
    0,.CURR$TEMP, 5, 0 */


/* EXCHANGE DEFINITION */
DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE
    (MESSAGE$HEAD               ADDRESS,
    MESSAGE$TAIL                ADDRESS,
    TASK$HEAD                   ADDRESS,
    TASK$TAIL                   ADDRESS,
    EXCHANGE$LINK               ADDRESS)';


/* ANALOG HANDLER AND RESPONSE EXCHANGES */
DECLARE RQAIEX      EXCHANGE$DESCRIPTOR EXTERNAL;
DECLARE RESP$EX     EXCHANGE$DESCRIPTOR;


/* DATA STORAGE */
DECLARE CURR$TEMP (5) ADDRESS;

/* POINTER TO RETURN MESSAGE */
DECLARE T1 ADDRESS;

/* EXTERNAL PROCEDURES */
RQCXCH: PROCEDURE (X) EXTERNAL;
    DECLARE X ADDRESS;
END RQCXCH;
RQSEND: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQSEND;
RQWAIT: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQWAIT;

/* BUILD RESPONSE EXCHANGE */
CALL RQCXCH(.RESP$EX);

/* SEND INPUT REQUEST, WAIT FOR RESPONSE */
CALL RQSEND(.RQAIEX,.REP$INP$MSG);
T1 = RQWAIT(.RESP$EX, 0);

/* USER ERROR CHECKING ROUTINE GOES HERE */
```

```
                EXTRN    RQAIEX, RQCXCH, RQSEND, RQWAIT

                DSEG
RESPEX:         DS       10     ;RESPONSE EXCHANGE
CTEMP:          DS       5      ;TEMPERATURE READING

INPMSG:                         ;REQUEST MESSAGE
                DS       2      ;LINK
                DS       2      ;LENGTH
                DS       1      ;TYPE
                DS       2      ;HOME EXCHANGE
                DS       2      ;RESPONSE EXCHANGE
                DS       2      ;STATUS
                DS       2      ;POINTER TO BASE REGISTER
                DS       1      ;CHANNEL-GAIN
                DS       1      ;NOT USED
                DS       2      ;POINTER TO DATA
                DS       2      ;COUNT
                DS       2      ;ACTUAL COUNT
                CSEG

; ASSUMED CONTENTS: 0, 21, 30, 0, RESPEX, 0,
                                ; 0F700H, 01011111B, 0, CRNTMP, 5, 0

; BUILD RESPONSE EXCHANGE
                LXI      B, RESPEX        ; ADDR WHERE EXCH IS TO BE BUILT
                CALL     RQCXCH           ; BUILD IT

; SEND  INPUT  REQUEST, WAIT  FOR  RESPONSE
                LXI      D,INPMSG     ; ADDR OF INPUT REQUEST MESSAGE
                LXI      B,RQAIEX     ; ADDR OF ANALOG INPUT EXCHANGE
                CALL RQSEND         ; SEND MESSAGE
                LXI      D,0          ; NO TIME LIMIT
                LXI      B,RESPEX     ; ADDR OF RESPONSE EXCHANGE
                CALL RQWAIT          ; WAIT

; USER ERROR CHECKING ROUTINE GOES HERE
```

Sequential Channel Input with Single Gain.

Request Message Format. A task requesting that a sequence of consecutive chan-
nels be sampled, with the same gain applied to each sample, creates and sends (to
RQAIEX) a message with the format shown in figure 8-3. Asterisks denote fields
that must be coded by the user task before the message is sent. Shaded fields are
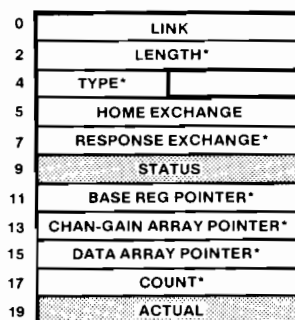those set or changed by the Analog Input Handler when it returns the message.
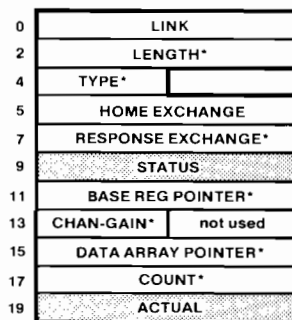
Figure 8-3. Request Message for Sequential Channel Input with Single Gain

LENGTH is 21.

TYPE is AISQS (31).

HOME EXCHANGE is not used by the Analog Input Handler.

RESPONSE EXCHANGE must specify the address of the user-defined exchange to which the Analog Input Handler is to send the message when the operation is completed.

When the Analog Input Handler returns this message, STATUS is set to 1, 2, 3, or 4, to indicate one of the following conditions:
1:   all data collected.
2:   (reserved for future use.)
3:   operation aborted; converter did not respond.
4:   (reserved for future use.)

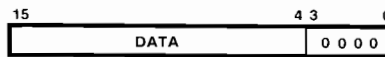BASE REG POINTER must specify the address of the Command-Status Register of the selected A/D converter.

CHAN-GAIN must specify the number of the first channel to be sampled and the gain that is to be applied to each sample. Its format is:

```
7     6  5  4                0
┌──────┬──┬──────────────────┐
│ GAIN │  │   CHANNEL NO.    │
└──────┴──┴──────────────────┘
```

CHANNEL NO., bits 0-4: a number 0-31.
GAIN, bits 6-7: a value of 0, 1, 2, or 3; represents a gain of x1, x2, x4, or x8, respectively.
Bit 5 is ignored.

DATA ARRAY POINTER must specify the address of the array to receive the sampled inputs. The number of elements in the array is equal to COUNT, and each element is two bytes. The Analog Input Handler will store the ith sample in the ith element of the array (recall that the ith element of the array has array index i-1), in the following format:

```
15                          4 3      0
┌───────────────────────────┬────────┐
│           DATA            │ 0 0 0 0 │
└───────────────────────────┴────────┘
```

Each element represents the actual value read from the A/D converter Data Register — 12 bits of data, followed by 4 zeros in the least significant bit positions. The voltage level corresponding to the 12-bit number is dependent upon the full-scale voltage range and the binary code, both of which are jumper-selected by the user as described in the Hardware Reference Manual for the analog input board. (See list of manuals at the beginning of this chapter.)

COUNT must specify the number of channels to be sampled. CHANNEL NO. plus COUNT must not exceed 32.

When the Analog Input Handler returns this message, ACTUAL is set to the actual number of channels sampled.

Example. The following coding example samples analog channels 0 through 31, which are attached to identical devices (or, at least, devices to whose output the same gain can be applied). The results are stored in the array CURR$TEMP; the gain applied is x2. This code could be used for paper factory roller temperature readings in the process control example discussed in Chapter 1 if it were expanded to include 32 identical rollers.

```
/* EXAMPLE OF SEQUENTIAL CHANNEL INPUT, SINGLE GAIN */

/* REQUEST MESSAGE */
DECLARE SQS$INP$MSG STRUCTURE
      (LINK                           ADDRESS,
      LENGTH                          ADDRESS,
      TYPE                            BYTE,
      HOME$EXCHANGE                   ADDRESS,
      RESPONSE$EXCHANGE               ADDRESS,
      STATUS                          ADDRESS,
      BASE$REG$POINTER                ADDRESS,
      CHAN$GAIN                       BYTE,
      UNUSED                          BYTE,
      DATA$ARRAY$POINTER              ADDRESS,
      COUNT                           ADDRESS,
      ACTUAL                          ADDRESS);
      /* ASSUMED CONTENT OF MESSAGES:*/
      /* 0, 21, AISQS, 0,.RESP$EX, 0, 0F700H, 01000000B,
      0,.CURR$TEMP, 32, 0 */


/* EXCHANGE DEFINITION */
DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE
      (MESSAGE$HEAD                   ADDRESS,
      MESSAGE$TAIL                    ADDRESS,
      TASK$HEAD                       ADDRESS,
      TASK$TAIL                       ADDRESS,
      EXCHANGE$LINK                   ADDRESS)';


/* ANALOG HANDLER AND RESPONSE EXCHANGES */
DECLARE RQAIEX        EXCHANGE$DESCRIPTOR EXTERNAL;
DECLARE RESP$EX       EXCHANGE$DESCRIPTOR;


/* DATA ARRAY */
DECLARE CURR$TEMP(32) ADDRESS;


/* POINTER TO RETURN MESSAGE */
DECLARE T1 ADDRESS;


/* EXTERNAL PROCEDURES */
RQCXCH: PROCEDURE (X) EXTERNAL;
    DECLARE X ADDRESS;
END RQCXCH;
RQSEND: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQSEND;
RQWAIT: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQWAIT;


/* BUILD RESPONSE EXCHANGE */
CALL RQCXCH(.RESP$EX);


/* SEND INPUT REQUEST, WAIT FOR RESPONSE */
CALL RQSEND(.RQAIEX,.SQS$INP$MSG);
T1 = RQWAIT(.RESP$EX, 0);


/* USER ERROR CHECKING ROUTINE GOES HERE */
```

Sequential Channel Input with Variable Gain.

Request Message Format. A task requesting that a sequence of consecutive chan-
nels be sampled, with a different gain applied to each sample, creates and sends (to
RQAIEX) a message with the format shown in figure 8-4. Asterisks denote fields
that must be coded by the user task before the message is sent. Shaded fields are
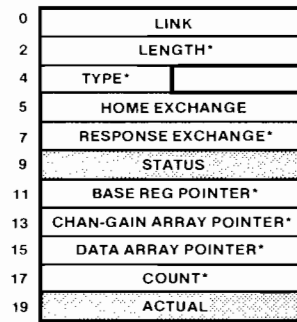those set or changed by the Analog Input Handler when it returns the message.

0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
11	BASE REG POINTER*
13	CHAN-GAIN ARRAY POINTER*
15	DATA ARRAY POINTER*
17	COUNT*
19	ACTUAL

**Figure 8-4. Request Message for Sequential Channel Input
with Variable Gain**

LENGTH is 21.

TYPE is AISQV (32).

HOME EXCHANGE is not used by the Analog Input Handler.

RESPONSE EXCHANGE must specify the address of the user-defined exchange to
which the Analog Input Handler is to send the message when the operation is com-
pleted.

When the Analog Input Handler returns this message, STATUS is set to 1, 2, 3, or 4,
to indicate one of the following conditions:
1:    all data collected.
2:    (reserved for future use.)
3:    operation aborted; converter did not respond.
4:    (reserved for future use.)

BASE REG POINTER must specify the address of the Command-Status Register of
the selected A/D converter.

CHAN-GAIN ARRAY POINTER must specify the address of the array giving the
input voltage gain to be applied to each sample. The gain in the ith position of this
array (recall that the ith element of the array has array index i-1) is applied to the ith
channel in the sequence. The first element in the array also specifies the channel
number of the first channel in the sequence. The number of elements in the array is
equal to COUNT, and each element is a byte formatted as follows:

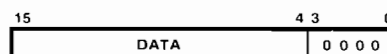| 7	6	5	4		0
GAIN			CHANNEL NO.		
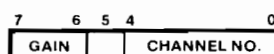
CHANNEL NO., bits 0-4: channel number of the first sample; a number 0-31.
Required only for the first element in the array, and must be set to zero in the
remaining elements.

GAIN, bits 6-7: gain to be applied to the ith sample; a value of 0, 1, 2, or 3; represents a gain of x1, x2, x4, or x8, respectively. Bit 5 is ignored.

DATA ARRAY POINTER must specify the address of the array to receive the sampled inputs. The number of elements in the array is equal to COUNT, and each element is two bytes. The Analog Input Handler will store the ith sample in the ith element of the array, in the following format:

```
15                                    4 3         0
┌──────────────────────────────────┬───────────┐
│              DATA                 │  0 0 0 0  │
└──────────────────────────────────┴───────────┘
```

Each element represents the actual value read from the A/D converter Data Register — 12 bits of data, followed by 4 zeros in the least significant bit positions. The voltage level corresponding to the 12-bit number is dependent upon the full-scale voltage range and the binary code, both of which are jumper-selected by the user as described in the Hardware Reference Manual for the analog input board. (See list of manuals an the beginning of this chapter.)

COUNT must specify the number of channels to be sampled. CHANNEL NO. plus COUNT must not exceed 32.

When the Analog Input Handler returns this message, ACTUAL is set to the actual number of channels sampled.

Example. The following coding example takes samplings from analog channels 30 and 31 and stores the values in structure DATA$ARRAY. A gain of x8 is applied to the input from channel 30, and a gain of x2 is applied to the input from channel 31. This code could be used to update both the roller pressure and roller temperature readings in the Chapter 1 process control example if these two operations were combined into one step.

```
/* EXAMPLE OF SEQUENTIAL CHANNEL INPUT, VARIABLE GAIN */

/* REQUEST MESSAGE */
DECLARE SQV$INP$MSG STRUCTURE
        (LINK                       ADDRESS,
        LENGTH                      ADDRESS,
        TYPE                        BYTE,
        HOME$EXCHANGE               ADDRESS,
        RESPONSE$EXCHANGE           ADDRESS,
        STATUS                      ADDRESS,
        BASE$REG$POINTER            ADDRESS,
        CHAN$GAIN$ARRAY$PTR         ADDRESS,
        DATA$ARRAY$POINTER          ADDRESS,
        COUNT                       ADDRESS,
        ACTUAL                      ADDRESS);
        /* ASSUMED CONTENT OF MESSAGE: */
        /* 0, 21, AISQV, 0,.RESP$EX, 0, 0F700H,.CG$ARRAY,
              .DATA$ARRAY, 2, 0 */
```

```
/* EXCHANGE DEFINITION */
DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE
    (MESSAGE$HEAD              ADDRESS,
    MESSAGE$TAIL               ADDRESS,
    TASK$HEAD                  ADDRESS,
    TASK$TAIL                  ADDRESS,
    EXCHANGE$LINK              ADDRESS)';


/* ANALOG HANDLER AND RESPONSE EXCHANGES */
DECLARE RQAIEX        EXCHANGE$DESCRIPTOR EXTERNAL;
DECLARE RESP$EX       EXCHANGE$DESCRIPTOR;


/* CHAN-GAIN ARRAY */
DECLARE CG$ARRAY (2) BYTE;
    /* ASSUMED CONTENTS: 11011110B, 01000000B */


/* DATA ARRAY (HERE DECLARED AS STRUCTURE IN ORDER
 * TO GIVE INDIVIDUAL NAMES TO ELEMENTS)*/
DECLARE DATA$ARRAY STRUCTURE
    (CURR$PRESS         ADDRESS,
    CURR$TEMP           ADDRESS);


/* POINTER TO RETURN MESSAGE */
DECLARE T1 ADDRESS;



/* EXTERNAL PROCEDURES */
RQCXCH: PROCEDURE (X) EXTERNAL;
    DECLARE X ADDRESS;
END RQCXCH;
RQSEND: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQSEND;
RQWAIT: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQWAIT;


/* BUILD RESPONSE EXCHANGE */
CALL RQCXCH(.RESP$EX);


/* SEND INPUT REQUEST, WAIT FOR RESPONSE */
CALL RQSEND(.RQAIEX,.SQV$INP$MSG);
T1 = RQWAIT(.RESP$EX, 0);


/* USER ERROR CHECKING ROUTINE GOES HERE */
```

Random Channel Input.

Request Message Format. A task requesting that a set of random channels be
sampled creates and sends (to RQAIEX) a message with the format shown in figure
8-5. Asterisks denote fields that must be coded by the user task before the message is

sent. Shaded fields are those set or changed by the Analog Input Handler when it returns the message.
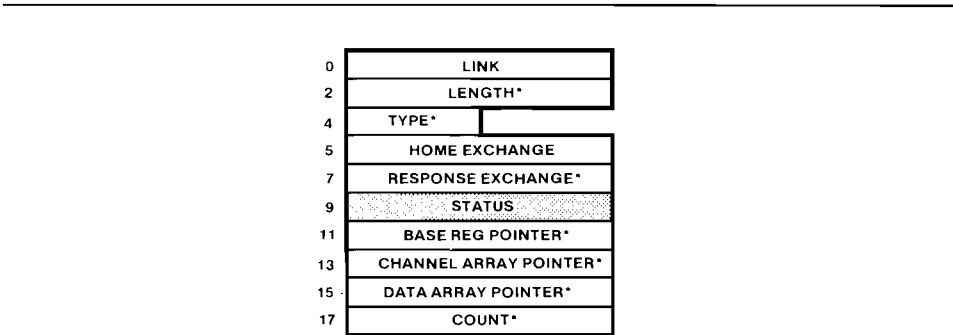
0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
11	BASE REG POINTER*
13	CHAN-GAIN ARRAY POINTER*
15	DATA ARRAY POINTER*
17	COUNT*
19	ACTUAL

Figure 8-5. Request Message for Random Channel Input

LENGTH is 21.

TYPE is AIRAN (33).

HOME EXCHANGE is not used by the Analog Input Handler.

RESPONSE EXCHANGE must specify the address of the user-defined exchange to which the Analog Input Handler is to send the message when the operation is completed.
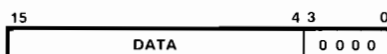
When the Analog Input Handler returns this message, STATUS is set to 1, 2, 3, or 4, to indicate one of the following conditions:
1:    all data collected.
2:    (reserved for future use.)
3:    operation aborted; converter did not respond.
4:    (reserved for future use.)

BASE REG POINTER must specify the address of the Command-Status Register of the selected A/D converter.

CHAN-GAIN ARRAY POINTER must specify the address of the array giving the channel number of each sample and the gain that is to be applied to it. The number of elements in the array is equal to COUNT, and each element is a byte formatted as. follows:

7 6 5 4	0
GAIN	CHANNEL NO.
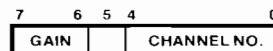
CHANNEL NO., bits 0-4: channel number of the ith sample; a number 0-31.
GAIN, bits 6-7: gain to be applied to the ith sample; a value of 0, 1, 2, or 3; represents a gain of x1, x2, x4, or x8, respectively.
Bit 5 is ignored.

DATA ARRAY POINTER must specify the address of the array to receive the sampled inputs. The number of elements in the array is equal to COUNT, and each element is two bytes. The Analog Input Handler will store the ith sample in the ith element of the array, in the following format:

15	4 3	0
DATA	0 0 0 0	

Each element represents the actual value read from the A/D converter Data Register — 12 bits of data, followed by 4 zeros in the least significant bit positions. The voltage level corresponding to the 12-bit number is dependent upon the full-scale voltage range and the binary code, both of which are jumper-selected by the user as described in the Hardware Reference Manual for the analog input board. (See list of manuals at the beginning of this chapter.)

COUNT must specify the number of channels to be sampled; the range is 1-254.

When the Analog Input Handler returns this message, ACTUAL is set to the actual number of samples collected.

Example. The following coding example is identical to the example given just previously (for sequential channel input with variable gain), except that the channels now being sampled are 0 and 31.

```
/* EXAMPLE OF RANDOM CHANNEL INPUT, VARIABLE GAIN */

/* REQUEST MESSAGE */
DECLARE RAN$INP$MSG STRUCTURE
      (LINK                        ADDRESS,
      LENGTH                       ADDRESS,
      TYPE                         BYTE,
      HOME$EXCHANGE                ADDRESS,
      RESPONSE$EXCHANGE            ADDRESS,
      STATUS                       ADDRESS,
      BASE$REG$POINTER             ADDRESS,
      CHAN$GAIN$ARRAY$PTR          ADDRESS,
      DATA$ARRAY$POINTER           ADDRESS,
      COUNT                        ADDRESS,
      ACTUAL                       ADDRESS);
      /* ASSUMED CONTENT OF MESSAGES */
      /* 0, 21, AIRAN, 0, .RESP$EX, 0, 0F700H, .CG$ARRAY,
      .DATA$ARRAY, 2, 0 */

/* EXCHANGE DEFINITION */
DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE
      (MESSAGE$HEAD                ADDRESS,
      MESSAGE$TAIL                 ADDRESS,
      TASK$HEAD                    ADDRESS,
      TASK$TAIL                    ADDRESS,
      EXCHANGE$LINK                ADDRESS)';

/* ANALOG HANDLER AND RESPONSE EXCHANGES */
DECLARE RQAIEX      EXCHANGE$DESCRIPTOR EXTERNAL;
DECLARE RESP$EX     EXCHANGE$DESCRIPTOR;

/* CHAN-GAIN ARRAY */
DECLARE CG$ARRAY (2) BYTE;
      /* ASSUMED CONTENTS: 11000000B, 01011111B */

/* DATA ARRAY HERE DECLARED AS STRUCTURE IN ORDER
   TO GIVE INDIVIDUAL NAMES TO ELEMENTS */
DECLARE DATA$ARRAY STRUCTURE
      (CURR$PRESS      ADDRESS,
      CURR$TEMP        ADDRESS);
```

```
/* POINTER TO RETURN MESSAGE */
DECLARE T1 ADDRESS;

/* EXTERNAL PROCEDURES */
RQCXCH: PROCEDURE (X) EXTERNAL;
    DECLARE X ADDRESS;
END RQCXCH;
RQSEND: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQSEND;
RQWAIT: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQWAIT;

/* BUILD RESPONSE EXCHANGE */
CALL RQCXCH(.RESP$EX);

/* SEND INPUT REQUEST, WAIT FOR RESPONSE */
CALL RQSEND(.RQAIEX,.RAN$INP$MSG);
T1 = RQWAIT(.RESP$EX, 0);

/* USER ERROR CHECKING ROUTINE GOES HERE */
```

Random Channel Output.

Request Message Format. A task requesting random channel output creates and sends (to RQAOEX) a message with the format shown in figure 8-6. Asterisks denote fields that must be coded by the user task before the message is sent. Shaded fields are those set or changed by the Analog Input Handler when it returns the message.

0	LINK
2	LENGTH*
4	TYPE*
5	HOME EXCHANGE
7	RESPONSE EXCHANGE*
9	STATUS
11	BASE REG POINTER*
13	CHANNEL ARRAY POINTER*
15	DATA ARRAY POINTER*
17	COUNT*

Figure 8-6. Request Message for Random Channel Output

LENGTH is 19.

TYPE is AORAN (38).

HOME EXCHANGE is not used by the Analog Output Handler.

RESPONSE EXCHANGE must specify the address of the user-defined exchange to which the Analog Input Handler is to send the message when the operation is completed.

When the Analog Input Handler returns this message, STATUS is set to 1 or 2, to indicate one of the following conditions:

1:    all data output.

2:    (reserved for future use.)

BASE REG POINTER must specify the address of the Holding Register for D/A Channel 0 of the converter receiving the outputs.

CHANNEL ARRAY POINTER must specify the address of the array giving the number of each channel to be set. The number of elements in the array is COUNT. Each element is a byte which may take on a value 0-3.

DATA ARRAY POINTER must specify the address of the array giving the digital values to be applied to the channels indicated in CHANNEL ARRAY. The number of elements in the array is equal to COUNT, and each element is a two-byte value having the following format:

```
15                              4 3       0
+-----------------------------+---------+
|            DATA             | 0 0 0 0 |
+-----------------------------+---------+
```

Each element represents the value to be output to the D/A converter registers — 12 bits of data, followed by 4 zeros in the least significant bit positions. The 12-bit number corresponding to a specific voltage level is dependent upon the full-scale voltage range and the binary code, both of which are jumper-selected by the user as described in the Hardware Reference Manual for the analog input board. (See list of manuals at the beginning of this chapter.)

COUNT must specify the number of channels to be set; the range is 1-4.

Example. The following coding example provides analog outputs on channels 2 and 3, respectively. This code could be used to make pressure and temperature adjustments in the process control example described in Chapter 1.

```
/* EXAMPLE OF RANDOM CHANNEL OUTPUT */

/* REQUEST MESSAGE */
DECLARE RAN$OUTP$MSG STRUCTURE
        (LINK                        ADDRESS,
        LENGTH                       ADDRESS,
        TYPE                         BYTE,
        HOME$EXCHANGE                ADDRESS,
        RESPONSE$EXCHANGE            ADDRESS,
        STATUS                       ADDRESS,
        BASE$REG$POINTER             ADDRESS,
        CHAN$GAIN$ARRAY$PTR          ADDRESS,
        DATA$ARRAY$POINTER           ADDRESS,
        COUNT                        ADDRESS);
        /* ASSUMED CONTENT OF MESSAGES:*/
        /* 0, 19, AORAN, 0,.OUT$RESP$EX, 0, 0F708H,.CH$ARRAY,
        .OUT$ARRAY, 2 */

/* EXCHANGE DEFINITION */
DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE
        (MESSAGE$HEAD                ADDRESS,
        MESSAGE$TAIL                 ADDRESS,
        TASK$HEAD                    ADDRESS,
        TASK$TAIL                    ADDRESS,
        EXCHANGE$LINK                ADDRESS)';
```

```
/* ANALOG HANDLER AND RESPONSE EXCHANGES */
DECLARE RQAOEX   EXCHANGE$DESCRIPTOR EXTERNAL;
DECLARE OUT$RESP$EX EXCHANGE$DESCRIPTOR;

/* CHANNEL ARRAY */
DECLARE CH$ARRAY (2) BYTE;
    /* ASSUMED CONTENTS: 2, 3 */

/* DATA ARRAY HERE DECLARED AS STRUCTURE IN ORDER
TO GIVE INDIVIDUAL NAMES TO ELEMENTS */
DECLARE OUT$ARRAY STRUCTURE
    (CORRECT$PRESS        ADDRESS,
    CORRECT$TEMP          ADDRESS);

/* POINTER TO RETURN MESSAGE */
DECLARE T1 ADDRESS;

/* EXTERNAL PROCEDURES */
RQCXCH: PROCEDURE (X) EXTERNAL;
    DECLARE X ADDRESS;
END RQCXCH;
RQSEND: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQSEND;
RQWAIT: PROCEDURE (X,Y) EXTERNAL;
    DECLARE (X,Y) ADDRESS;
END RQWAIT;

/* BUILD RESPONSE EXCHANGE */
CALL RQCXCH(.OUT$RESP$EX);

/* SEND OUTPUT REQUEST, WAIT FOR RESPONSE */
CALL RQSEND(.RQAOEX,.RAN$OUTP$MSG);
T1 = RQWAIT(.OUT$RESP$EX, 0);

/* USER ERROR CHECKING ROUTINE GOES HERE */
```

Configuration, Linking, and Locating

This section supplies the information you need to include the Analog Input Handler and/or Analog Output Handler in your configuration module, and to link and locate the resulting object code. Instructions for preparing the configuration module from this information are given in Chapter 3, along with general instructions for linking and locating your system.

Configuration Requirements

Analog Input Handler.

Tasks. One task - RQAIH - must be defined. The stack length must be 34, and the default exchange is RQAIEX. The priority may be any level that is appropriate within your system.

Initial Exchanges. You must define one initial exchange, RQAIEX. One or more user exchanges must also be defined to receive responses from the Analog Input Handler, unless your tasks create them at run time via RQCXCH.

PUBLIC Data. No PUBLIC data items need be defined at configuration time for the Analog Input Handler.

Analog Output Handler.

Tasks. One task - RQAOH - must be defined. The stack length must be 32, and the default exchange is RQAOEX. The priority may be any level that is appropriate within your system.

Initial Exchanges. You must define one initial exchange, RQAOEX. One or more user exchanges must also be defined to receive responses from the Analog Output Handler, unless your task creates them at run time via RQCXCH.

PUBLIC Data. No PUBLIC data items need be defined at configuration time for the Analog Output Handler.

Linking and Locating

To link the Analog Input Handler to your application code, add the library file AIHDLR.LIB to the list of input files given to the LINK program. To link in the Analog Output Handler, add the file AOHDLR.LIB.

No special information is needed to locate object code for a system that includes the Analog Handlers; for general instructions, refer to Chapter 3.

General Description

The RMX/80 Bootstrap Loader is an extension to RMX/80 that adds considerable flexibility to iSBC hardware systems equipped with disk. The purpose of the Bootstrap Loader is to allow you to store your applications software on disk rather than in ROM. This affords you two types of flexibility:

- Since your applications software is stored on disk rather than in ROM, changes are easier to implement. A modification to your applications software entails changing the contents of the disk rather than replacing the ROM.

- Because the disk has a much larger storage capacity than does the ROM, you can store multiple applications on disk and use the Bootstrap Loader to bring them into RAM as they are needed. This feature allows you to use a single iSBC for more than one application.

Functions Provided

In order to understand the capabilities provided by the Bootstrap Loader, you should first know something about its implementation. The Loader is implemented as a task that runs under RMX/80. Should you decide to use the Loader, you must include it in the system that you burn into ROM. Throughout this chapter, any ROM-resident system that includes the Bootstrap Loader is called a Loader System.

The Bootstrap Loader provides two capabilities: bootstrap loading and program controlled loading. Bootstrap loading is invoked when you send a RESET signal to your iSBC. This action causes a file, the name of which is a parameter burned into ROM along with the Loader System, to be loaded from Drive 0 into RAM. Program controlled loading allows the program to specify the name of the file to be loaded and to invoke the loading operation.

Throughout this chapter, files to be loaded by the Bootstrap Loader are called loadable files. The content of such files is flexible, but the form is not. Loadable files can contain tasks, exchanges, data, taskless code segments or a combination of any of these. However, regardless of its content, the file must be absolutely located. In other words, the file must be of the same form as files produced by the ISIS-II LOCATE command.

Whenever a file is loaded, two events occur. First, the Loader reads the file from Drive 0 into RAM. Second, a new system is created. Called the Loaded System, it contains all the tasks, exchanges and code segments from the Loader System as well as those read from the disk.

Software Environment

The Bootstrap Loader is a task that runs under RMX/80. Although the Loader does access the disk, it does not require the entire Disk File System. The following software components are needed to support this task:

- the RMX/80 Nucleus

- the DISKIO Module of the Disk File System

- the INTEL-supplied, Disk Controller Module used to interface the DISKIO Module with your iSBC disk controller

- the INTEL-supplied VECRST Module used to initialize the system that supports the Loader

- your configuration module

- any additional exchanges, applications tasks or RMX/80 extensions that you wish to place in ROM

It is these components which comprise the Loader System, and it is this system that is stored in ROM instead of your applications software.

Hardware Environment

In order for an iSBC installation to be able to use the RMX/80 Bootstrap Loader, the installation must include the following hardware:

- an iSBC 80/10, 80/10A, 80/20, 80/20-4 or 80/30

- at least 4K bytes of PROM to contain the Loader System

- a sufficient amount of RAM to contain the system or systems to be loaded

- an additional 1000 bytes of RAM (of which at least 336 bytes must be controller addressable) for use by the Loader System

- an iSBC disk controller board

- Drive 0

- a disk

Using the RMX/80 Bootstrap Loader

The RMX/80 Bootstrap Loader supports both bootstrap loading and program controlled loading. The following paragraphs provide instructions for using the Loader for either purpose. Regardless of the type of loading performed, once the Loaded System has been created, RMX/80 assigns the processor to the ready task having the highest priority.

Bootstrap Loading

Bootstrap loading can be manually invoked by either of two methods. The operator can supply power to the iSBC or, if power has already been supplied, the operator can otherwise cause a RESET signal to be sent to the iSBC. In either case, the Loader System will be restarted and control will be given to the ready task having the highest priority.

When the Bootstrap Loader Task becomes the ready task having the highest priority, it will receive control and will attempt to read the previously specified file from Drive 0. If the loading attempt is not successful, the Loader will make repeated attempts at five second intervals until a successful loading operation has been achieved.

Program Controlled Loading

Program controlled loading can be accomplished by either of two methods. The first method allows the invoking program to select the file to be loaded. There are two steps involved in invoking this type of loading:

1) The invoking program must specify the name of the file to be loaded. This is accomplished by modifying the two-byte, Loader System PUBLIC variable called RQNAME so that it points to a nine-byte array containing the name of the file to be loaded. This file name must conform to a specific format. The leftmost six bytes are reserved for the name of the file, and the rightmost three bytes are reserved for the extension. Both the file name and the extension must be left-justified within their respective fields, and any unused bytes must be filled with ASCII null characters (0H). Other conventions used in naming files can be found in Chapter 7.

2) Control must be transferred to RQSTRT, a Loader System PUBLIC symbol. This is the starting address of the START Module. Once this module has received control, the system will, with one difference, behave as through it were performing a bootstrap loading operation. The one difference is that the program specifies the name of the file to be loaded.

The second method of program controlled loading reads the same file as is read by the bootstrap loading process. To invoke this type of loading, you need only transfer control to location 0H. The resultant behavior is in all ways identical to that which occurs during bootstrap loading.

Implementing a Loader System

There are two procedures which you must follow in order to implement a Loader System. The first procedure is used to create the PROM containing the Loader System, and the second is used to create disk files that can be loaded.

Creating a Loader System PROM

The process of creating a PROM containing a Loader System is comprised of four major steps:

1) Configuring the Loader System
2) Linking the components of the Loader System
3) Locating the components of the Loader System
4) Burning the PROM

Steps 1 through 3 are discussed in detail below. Step 4, however, is no different for the Loader System than for any other system. Therefore, the processes involved in burning the PROM are not discussed in this chapter. A discussion of PROM burning can be found in both the INTELLEC 800 Microcomputer Development System Operator's Manual, 9800129, and the Universal PROM Programmer Manual, 9800133.

Configuring the Loader System. As with any RMX/80 system, you must select or, in some cases, create the components of your Loader System. Each of the required components is described below. The following five components are supplied with RMX/80:

- VECRST Module

  The purpose of this module is to reroute some of the RST interrupt traps from ROM to RAM. Were this not done, any of the tasks loaded from disk into RAM would not have access to RST interrupts 1, 2, 3, 4, 5, 6, and 7. By incorporating the VECRST Module into the Loader System, the RST interrupts listed above are rerouted to location RQRSTV + 8n, where RQRSTV is a name declared PUBLIC by the INTEL-supplied loader library and n is the RST interrupt number.

  Only the listed RST interrupts are affected by this module. The RST 0 interrupt is reserved as the RESET interrupt and can be used only to restart the Loader System. Consequently, your RAM-resident tasks cannot redefine the RST 0 interrupt.

  If your hardware system uses an iSBC 80/30, it is equipped with an 8085A microprocessor and has four additional interrupts: TRAP and RST 5.5, 6.5 and 7.5. Although these interrupts are not rerouted to RAM through the VECRST Module, your RAM-resident tasks can connect with them through the standard RMX/80 RQSETV Nucleus operation.

- DISKIO Module

  This module is one of two components of the Disk File System needed by the Loader Task. The capabilities provided by the DISKIO Module are discussed in Chapter 7.

- Disk Controller Software Module

  This module is the other component of the Disk File System required by the Loader Task. It allows the DISKIO Module to communicate with the disk controller hardware. The method of incorporating the correct module into the Loader System is discussed in Chapter 7.

- Loader Library

  This INTEL-supplied library contains the software associated with the Loader Task.

- RMX/80 Nucleus

  This is, of course, the heart of the Loader System.

You must supply the following two components:

- Configuration Module

  Each task or exchange to be stored in ROM must have an entry in the Create Table, RQCRTB. Consequently, if you wish to place applications tasks or exchanges in the Loader System, they too must have entries in RQCRTB. Since the following discussion is limited to entries made only on behalf of the Loader Task, you may wish to refer to the sections of Chapters 3 and 7 that provide additional information.

  The Loader Task requires a Static Task Descriptor containing the following information:

  — NAME

    You may select the name of the Loader Task.

  — INITIAL P.C.

    The initial program counter of the Loader Task should be set to RQBOOT, a value declared PUBLIC in the INTEL-supplied loader library.

— STACK LENGTH

   The Loader Task requires at least 64 bytes.

— PRIORITY

   Unless you wish to have some task invoked before the Loader, you should
   assign a priority higher than that assigned to any other task not driven
   by interrupts.

— DEFAULT EXCHANGE

   The Loader does not need a default exchange. Consequently, the default
   exchange should be set to 0.

The Static Task Descriptor can be created using an STD macroinstruction, as
in the following example:

<div align="center">STD RQBOOT,64,129,0</div>

If you wish to specify the name of the file to be loaded when the RESET signal is
received, the configuration module is the place to do it. The name must be
contained in a nine-byte array called RQFILE, and the array must be declared
PUBLIC. Additional rules for encoding the name of the file can be found in the
section of this chapter entitled "Program Controlled Loading."

If you do not specify the name of the file, the Loader Task will use the default
file name, RMXSYS, which is supplied by BOTUNR.LIB.

• Controller Addressable RAM Module

   The Loader Task expects a 256-byte buffer, called RQPOOL, to be located in
   RAM that can be accessed by the disk controller. If your hardware system uses
   an iSBC 80/10 or 80/20, you must define this buffer as a PUBLIC variable
   within the controller addressable RAM module. Even if your hardware system
   uses an iSBC 80/30, you still must name the buffer RQPOOL and you still must
   declare it PUBLIC. The RQPOOL buffer can be used by other tasks when the
   Loader is inactive.

   You should use the information contained in Chapter 7 when constructing this
   module.

In addition to these required components, you optionally may add other RMX/80
extensions as well as applications tasks and exchanges.

Linking the Loader System. The following example of an ISIS LINK command
can be used to link your Loader System:

```
LINK  :Fn:BOT8x0.LIB(VECRST),&
      :Fn:RMX8x0.LIB(START),&
      :Fn:USER.LIB,&
      :Fn:BOT8x0.LIB,&
      :Fn:DIO8x0.LIB,&
      :Fn:DFSUNR.LIB,&
      :Fn:CAMMOD.OBJ,&
      :Fn:RMX8x0.LIB,&
      :Fn:BOTUNR.LIB,&
      :Fn:PLM80.LIB TO :Fn:BOTSYS.LNK
```

where x is either 1, 2 or 3 depending upon whether your iSBC is an 80/10, 80/20 or
80/30, respectively. The order in which the libraries and modules are linked is
significant.

The contents of each library are briefly described below:

- BOT8x0.LIB

  This is the INTEL-supplied library that contains the modules relating directly to the Loader Task.

- RMX8x0.LIB

  This library is also supplied by INTEL. It contains the RMX/80 Nucleus as well as a special system initialization module called START.

- USER.LIB

  You must create this library. It contains the configuration module as well as any tasks and exchanges you wish to incorporate in the Loader System.

- DIO8x0.LIB

  This is the INTEL-supplied library containing the elements of the Disk File System needed by the Loader System. If your hardware system uses an iSBC 80/10, you can save about 50 bytes of ROM by calling out the specific modules needed from this library. The names of the specific modules depend upon the disk controller board used in conjunction with your iSBC 80/10. If you are using a Model 201 or 202 disk controller board, replace the expression

    :Fn:DIO8x0.LIB

  with the following:

    :Fn:DIO810.LIB(DISKIO,HAN212,V10HD1)

  If, on the other hand, you are using a Model 204 disk controller board, replace the same expression with the following:

    :Fn:DIO810.LIB(DISKIO,HAN204,V10HD4)

  Using either of these memory-saving substitutions causes the LINK program to issue a warning about an unresolved external reference to either RQHD1V (for the 204 board) or RQHD4V (for the 201 or 202 boards). The unresolved reference will not cause any problems as it refers to a polling routine that is not used by the Loader Task.

- DFSUNR.LIB

  This INTEL-supplied library is used to provide default assignments to Disk File System external references which, without this library, would be unresolved.

- CAMMOD.OBJ

  This object module is the controller addressable RAM module discussed both in the section of this chapter entitled "Configuring the Loader System" and in Chapter 7. You must create this module; INTEL does not supply it.

- BOTUNR.LIB

  This INTEL-supplied library is used to provide default assignments to any Loader System external references which, without this library, would be un-resolved.

- PLM80.LIB

  This INTEL-supplied library contains subroutines used by the Loader System.

In addition to the object module and the libraries listed above, you can link any other modules that you wish to include in the Loader System. For example, if your hardware system uses an iSBC 80/10 with an attached iSBC 104 to supply an external clock, you will need to link the SBC104.LIB before you link the UNRSLV.LIB.

Locating the Loader System. The following ISIS LOCATE command can be used to assign memory to the Loader System:

    LOCATE   :Fn:BOTSYS.LNK TO :Fn:BOTSYS&
                   CODE(40) DATA (xxxxH) STACKSIZE(0)

The code segment must be located at location 40H and the stack size parameter must be set to zero. The data segment can be placed in RAM wherever you wish.

Creating a Loadable System

The processes involved in creating a system that can be loaded by the RMX/80 Bootstrap Loader are, with two exceptions, identical to the processes used to create any other RMX/80 system. The two exceptions are as follows:

- Main Program Module

  In systems not loaded from the disk, the main program module is the START Module. The purpose of the START Module is to construct the system from information contained in the configuration module. In systems loaded from disk, a similar function is performed by the LODINI Module. The difference between LODINI and START is that the LODINI Module preserves those components of the Loader System which are located in RAM, whereas the START Module does not.

- Loader PUBLIC Definitions

  The loaded system must be linked to the PUBLIC definitions of the Loader System if the new tasks are to have access to the components of the Loader System.

Configuring a Loadable System. The following libraries are needed to create a loadable system:

- LOD8x0.LIB

  This is the INTEL-supplied library containing the LODINI Module and other code segments needed to start the system after it is loaded.

- USRCOD.LIB

  This library represents the object modules that you must supply. They can be supplied as object modules, libraries, a combination of both or as a single library. The collection can have any name you wish, but it must contain a configuration module and your applications modules. The presence of RQCRTB,

the Create Table, in the configuration module is required even though it will cause the LINK program to generate a warning about a multiple definition.

NOTE

Tasks should be placed in the Create Table with higher priority tasks preceding lower priority tasks. If this is not done and an interrupt driven task is activated while the Loader Task is running, the running task might attempt to wait at an exchange which has not yet been created.

- Other RMX/80 Libraries

  You can incorporate into the system any RMX/80 extensions required by your application tasks. Even without adding any RMX/80 extensions, however, your tasks can use the Nucleus and DISKIO capabilities.

- UNRSLV.LIB

  This INTEL-supplied library is used to provide default assignments for RMX/80 unresolved external references.

- PLM80.LIB

  This INTEL-supplied library contains subroutines commonly used by programs written in PLM/80.

Linking a Loadable System. You can use the following LINK command to link your loadable system:

```
LINK  :Fn:LOD8x0.LIB(LODINI),&
      :Fn:USRCOD.LIB,&
      :Fn:LOD8x0.LIB,&

            .
            .
            .
      (Any other RMX/80 extensions desired)
            .
            .
            .
      PUBLICS(:Fn:BOTSYS),&
      :Fn:RMX8x0.LIB,&
      :Fn:UNRSLV.LIB,&
      :Fn:PLM80.LIB TO :Fn:RMXSYS.LNK
```

The order in which the files are linked is significant.

During the linking process, two warnings about multiple definitions may appear. One will mention RQCRTB, and, if the active Debugger is linked in, a second one will mention R?RST5HD. Neither of these is cause for concern. However, no other multiple definitions should occur unless the loaded system redefines tasks, exchanges, PUBLIC symbols or interrupts used in the Loader System. Should any interrupts be redefined, the loaded system is responsible for re-enabling the duplicated interrupt level. The can be done with the RQELVL Nucleus operation.

Locating the Loadable System. The loadable system can be located by using the following ISIS LOCATE command:

```
LOCATE  :Fn:RMXSYS.LNK TO :Fn:RMXSYS&
        CODE(xxxxH) STACKSIZE(0)
```

In the preceding example, the name of the file receiving the loadable system is RMXSYS. This file name can be anything you desire so long as it conforms to the file naming conventions discussed in Chapter 7. The name you select, however, is the name that must be used by the Loader Task when it attempts to load the file.

The code segment can be located in any RAM not used by the Loader System.

This appendix provides a complete list of all the files supplied on the RMX/80 product diskettes, with a brief description of the contents of each. The files are grouped here first according to diskette (Executive or Extensions), then (within each of these two groups) according to type of file.

Executive Diskette —
iSBC 80/20, 80/30, 80/10 Versions

The following files reside on the iSBC 80/20, 80/30, and 80/10 versions of the RMX/80 Executive diskette. These diskettes contain all files required by the RMX/80 Nucleus, Terminal Handler, Free Space Manager, Debugger, and Demonstration System, plus the basic assembly language macros for use in the configuration module. Some files are particular to one or two of the three versions; where this is true, the file description indicates which version(s) of the Executive diskette include that file.

Relocatable Object Code Files

RMX820.LIB:	Modules for the RMX/80 Nucleus (80/20 diskette only).
RMX830.LIB:	Modules for the RMX/80 Nucleus (80/30 diskette only).
RMX810.LIB:	Modules for the RMX/80 Nucleus (80/10 diskette only).
THI820.LIB:	Modules for the input side of the full Terminal Handler (80/20 diskette only).
THI830.LIB:	Modules for the input side of the full Terminal Handler (80/30 diskette only).
THI810.LIB:	Modules for the input side of the full Terminal Handler (80/10 diskette only).
THO820.LIB:	Modules for the output side of the full Terminal Handler (80/20 diskette only).
THO830.LIB:	Modules for the output side of the full Terminal Handler (80/30 diskette only).
THO810.LIB:	Modules for the output side of the full Terminal Handler (80/10 diskette only).
MTI820.LIB:	Modules for the input side of the minimal Terminal Handler (80/20 diskette only).
MTI830.LIB:	Modules for the input side of the minimal Terminal Handler (80/30 diskette only).
MTI810.LIB:	Modules for the input side of the minimal Terminal Handler (80/10 diskette only).

MTO820.LIB:	Modules for the output side of the minimal Terminal Handler (80/20 diskette only).
MTO830.LIB:	Modules for the output side of the minimal Terminal Handler (80/30 diskette only).
MTO810.LIB:	Modules for the output side of the minimal Terminal Handler (80/10 diskette only).
TSK820.LIB:	Modules for the Free Space Manager (80/20 diskette only).
TSK830.LIB:	Modules for the Free Space Manager (80/30 diskette only).
TSK810.LIB:	Modules for the Free Space Manager (80/10 diskette only).
ADB820.LIB:	Modules for the active portion of the Debugger (80/20 diskette only).
ADB830.LIB:	Modules for the active portion of the Debugger (80/30 diskette only).
ADB810.LIB:	Modules for the active portion of the Debugger (80/10 diskette only).
PDB820.LIB:	Modules for the passive portion of the Debugger (80/20 diskette only).
PDB830.LIB:	Modules for the passive portion of the Debugger (80/30 diskette only).
PDB810.LIB:	Modules for the passive portion of the Debugger (80/10 diskette only).
UNRSLV.LIB:	Declarations for resolving references to RMX/80-defined public symbols not declared in the application code.
PLM80.LIB:	PL/M support procedures required by the RMX/80 Nucleus and extensions.

PL/M INCLUDE Files

The PL/M source code files listed in this section are useful when coding calls to RMX/80 in PL/M. To include any of these sections of code in your program, you need not write out the code — simply provide the name of the appropriate file in a PL/M $INCLUDE command.

The following files contain PL/M declarations of various RMX/80 data structure formats and related symbolic constants. The most-used of these files are printed here for your convenience.

8020.ELT:	Declarations of various symbolic values relating to the iSBC 80/20. Most of these values are not needed when using RMX/80, particularly when the Terminal Handler is used. They are provided for reference and for possible user extensions, such as the use of counter 1 of the 8253 Programmable Interval Timer. (Provided on 80/20 diskette only.)

8030.ELT: Declarations of various symbolic values relating to
                             the iSBC 80/30. Most of these values are not need-
                             ed when using RMX/80, particularly when the
                             Terminal Handler is used. They are provided for
                             reference and for possible user extensions, such as
                             the use of counter 1 of the 8253 Programmable In-
                             terval Timer. (Provided on 80/30 diskette only.)

8010.ELT: Declarations of symbolic values associated with
                             the 8251 Programmable Communications Inter-
                             face (USART) to assist the user in any custom
                             software in which the USART plays a part.
                             (Provided on 80/10 diskette only.)

INTRPT.ELT: Declarations of symbolic values for the 8085
                             on-chip interrupts TRAP, ALEV, BLEV, and
                             C$LEV. (Provided on 80/30 diskette only).

MSGTYP.ELT: Declarations of RMX/80-defined message types.
                             (See Appendix C.)

STATIC.ELT: Declaration of Static Task Descriptor.

```
DECLARE STATIC$TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
           NAME(6)              BYTE,
           PC                   ADDRESS,
           SP                   ADDRESS,
           STKLEN               ADDRESS,
           PRIORITY             BYTE,
           EXCHANGE$ADDRESS     ADDRESS,
           TASK$PTR             ADDRESS)';

DECLARE CREATE$TABLE LITERALLY 'STRUCTURE (
           TASK$POINTER         ADDRESS,
           TASK$COUNT           BYTE,
           EXCHANGE$POINTER     ADDRESS,
           EXCHANGE$COUNT       BYTE)';
```

TASK.ELT: Declaration of Task Descriptor.

```
DECLARE TASK$DESCRIPTOR LITERALLY 'STRUCTURE (
           DELAY$LINK$FORWARD   ADDRESS,
           DELAY$LINK$BACK      ADDRESS,
           THREAD               ADDRESS,
           DELAY                ADDRESS,
           EXCHANGE$ADDRESS     ADDRESS,
           SP                   ADDRESS,
           MARKER               ADDRESS,
           PRIORITY             BYTE,
           STATUS               BYTE,
           NAME$PTR             ADDRESS,
           TASK$LINK            ADDRESS)';
DECLARE BOTTOM$FLAG LITERALLY '0C7C7H',
           UNUSED$FLAG LITERALLY '0C7H';
DECLARE TD$DELAY$LINK$FORWARD$OFFSET LITERALLY '0';
DECLARE TD$THREAD$OFFSET LITERALLY '4';
DECLARE TD$TASK$LINK$OFFSET LITERALLY '18';
DECLARE TASK$DESCRIPTOR$LENGTH LITERALLY '20';
```

EXCH.ELT: Declaration of Exchange Descriptor.


```
DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        MESSAGE$HEAD            ADDRESS,
        MESSAGE$TAIL            ADDRESS,
        TASK$HEAD              ADDRESS,
        TASK$TAIL              ADDRESS,
        EXCHANGE$LINK           ADDRESS)';
DECLARE ED$EXCHANGE$LINK$OFFSET LITERALLY '8';
DECLARE EXCHANGE$DESCRIPTOR$LENGTH LITERALLY '10';
```


IED.ELT: Declaration of Interrupt Exchange Descriptor.


```
DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        MESSAGE$HEAD            ADDRESS,
        MESSAGE$TAIL            ADDRESS,
        TASK$HEAD              ADDRESS,
        TASK$TAIL              ADDRESS,
        EXCHANGE$LINK           ADDRESS,
        LINK                  ADDRESS,
        LENGTH                ADDRESS,
        TYPE                  BYTE)';
```

MSG.ELT: Declaration of message.

```
DECLARE MSG$HDR LITERALLY '
        LINK                  ADDRESS,
        LENGTH                ADDRESS,
        TYPE                  BYTE,
        HOME$EXCHANGE          ADDRESS,
        RESPONSE$EXCHANGE     ADDRESS';
DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE (
        MSG$HDR,
        REMAINDER(1)           BYTE)';
DECLARE MSG$LINK$OFFSET LITERALLY '0';
DECLARE MIN$MSG$LENGTH LITERALLY '5';
```


THMSG.ELT: Declaration of Terminal Handler read or write
                                  request message.

```
DECLARE TH$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS                ADDRESS,
        BUFFER$ADR             ADDRESS,
        COUNT                 ADDRESS,
        ACTUAL                ADDRESS,
        REMAINDER(128)         BYTE)';
DECLARE MIN$TH$MSG$LENGTH LITERALLY '17';
```


COMMON.ELT: Declarations of common symbolic constants.


```
DECLARE TRUE LITERALLY '0FFH';
DECLARE FALSE LITERALLY '00H';
DECLARE BOOLEAN LITERALLY 'BYTE';
DECLARE FOREVER LITERALLY 'WHILE 1';
```

CHAR.ELT: Declarations of various non-printable ASCII characters.

```
DECLARE
        NULL            LITERALLY '00H',
        CONTROL$C       LITERALLY '03H',
        CONTROL$E       LITERALLY '05H',
        BELL            LITERALLY '07H',
        TAB             LITERALLY '09H',
        LF              LITERALLY '0AH',
        VT              LITERALLY '0BH',
        FF              LITERALLY '0CH',
        CR              LITERALLY '0DH',
        CONTROL$P       LITERALLY '10H',
        CONTROL$Q       LITERALLY '11H',
        CONTROL$R       LITERALLY '12H',
        CONTROL$S       LITERALLY '13H',
        CONTROL$X       LITERALLY '18H',
        CONTROL$Z       LITERALLY '1AH',
        ESC             LITERALLY '1BH',
        QUOTE           LITERALLY '22H',
        LCA             LITERALLY '61H',
        LCZ             LITERALLY '7AH',
        RUBOUT          LITERALLY '7FH';
```

The following files contain PL/M declarations for RMX/80 procedures, exchanges, etc.:

SYNCH.EXT: Declarations of the RQSEND, RQWAIT, RQACPT, and RQISND procedures.

RDYLST.EXT: Declaration of the RQACTV variable.

INTRPT.EXT: Declarations of the RQELVL, RQDLVL, RQSETV, and RQENDI procedures.

DLTASK.EXT: Declaration of the RQDTSK procedure.

DLEXCH.EXT: Declaration of the RQDXCH procedure.

RESUME.EXT: Declaration of the RQRESM procedure.

SUSPND.EXT: Declaration of the RQSUSP procedure.

OBJMAN.EXT: Declarations of the RQCTSK and RQCXCH procedures.

FSMGR.EXT: Declarations of the Free Space Manager allocation and reclamation exchanges.

THDINI.EXT: Declarations of the Terminal Handler input exchanges.

THDINO.EXT: Declarations of the Terminal Handler output exchanges.

Assembly Language INCLUDE Files

The assembly language source code files listed in this section are useful when coding

the RMX/80 configuration module in assembly language. To include any of these sections of code in your program, you need not write out the code — simply provide the name of the appropriate file in an assembler $INCLUDE command.

STD.MAC: Code for the macro STD.

XCHADR.MAC: Code for the macro XCHADR.

GENTD.MAC: Code for the macro GENTD.

CRTAB.MAC: Code for the macro CRTAB.

Demonstration System Files

The following files are provided to support the RMX/80 Demonstration System, which is described in Appendix H.

DEMO.LIB: Relocatable object modules for the Demonstration System tasks.

SBC104.LIB: Relocatable object modules provided for manipulating the clock hardware on the iSBC 104, 108, 116, and 517 combination memory and I/O expansion boards. (Provided on 80/10 diskette only.)

DEMO.M80:   Assembly language source code for the Demonstration System configuration module.

DEMO.PLM: PL/M source code for the Demonstration System configuration module.

GET.PLM: PL/M source code for the Demonstration System Free Space Manager Exerciser module, which contains the routines for the commands GET, PUT, POOL, and MESSAGE.

CLI.PLM: PL/M source code for the Demonstration System CLI (Command Line Interpreter) module.

SBC104.PLM: PL/M source code for the modules provided for manipulating the clock hardware on the iSBC 104, 108, 116, and 517 boards. (Provided on 80/10 diskette only.)

CLI.EXT: External declarations for the PUBLIC symbols provided by the CLI module.

SCANIN.PEX: External declaration of the utility procedure SCANIN.

NUMOUT.PEX: External declaration of the utility procedure NUMOUT.

DBLANK.PEX: External declaration of the utility procedure DBLANK.

DLIMIT.PEX:	External declaration of the utility procedure DLIMIT.
SEQ.PEX:	External declaration of the utility procedure SEQ.
DEMO.CSD:	ISIS-II SUBMIT file used to generate the Demonstration System.
LOADX.CSD:	ISIS-II SUBMIT file used to load the Demonstration System with ICE-80 or ICE-85.
DEMO:	Absolute object code (LINKed and LOCATEd) for the Demonstration System.

Extensions Diskette

The following files reside on the RMX/80 Extensions diskette. This diskette contains all files specific to the Disk File System (DFS) and the Analog Handlers.

Relocatable Object Code Files

DFSDIR.LIB:	Modules to implement the DFS directory structure.
DIO820.LIB:	Basic DFS I/O routines for the iSBC 80/20.
DIO830.LIB:	Basic DFS I/O routines for the iSBC 80/30.
DIO810.LIB:	Basic DFS I/O routines for the iSBC 80/10.
DFSUNR.LIB:	Declarations for resolving references to DFS-defined public symbols not declared in the application code.
AIHDLR.LIB:	Modules for the Analog Input Handler.
AOHDLR.LIB:	Modules for the Analog Output Handler.
BOT810.LIB:	Bootstrap Loader System modules for the iSBC 80/10.
BOT820.LIB:	Bootstrap Loader System modules for the iSBC 80/20.
BOT830.LIB:	Bootstrap Loader System modules for the iSBC 80/30.
LOD810.LIB:	Modules to be linked to systems loaded by the Bootstrap Loader System for the iSBC 80/10.
LOD820.LIB:	Modules to be linked to systems loaded by the Bootstrap Loader System for the iSBC 80/20.
LOD830.LIB:	Modules to be linked to systems loaded by the Bootstrap Loader System for the iSBC 80/30.
BOTUNR.LIB:	Declarations for resolving references to RMX-defined PUBLIC symbols not declared in the applications code of a Bootstrap Loader System.

PL/M INCLUDE Files

The PL/M source code files listed in this section are useful when using PL/M to code applicable tasks that interface with the Disk File System or the Analog Handlers. To include any of these sections of code in your program, you need not write out the code — simply provide the name of the appropriate file in a PL/M $INCLUDE command.

The following files contain declarations needed to build a PL/M configuration module for DFS systems. Each of the first seven files (ATTRIB.CFG through RENAME.CFG) contains the exchange associated with one or more DFS services and the intial program counter(s) of the routine(s) that perform the service(s). These parameters are used in the Initial Exchange Table and the Initial Task Table. The files do not include a stack length.

ATTRIB.CFG: Declarations for ATTRIB service.

DELETE.CFG: Declarations for DELETE service.

DIRECT.CFG: Declarations for directory services (OPEN, READ, WRITE, and CLOSE).

DISKIO.CFG: Declarations for DISKIO service.

FORMAT.CFG: Declarations for FORMAT service.

LOAD.CFG: Declarations for LOAD service.

RENAME.CFG: Declarations for RENAME service.

FSMGR.CFG: Declarations of Free Space Manager exchanges (needed for dynamic allocation of buffers).

HAND1.CFG: Declarations of initial program counter for controller task code for iSBC 201 and 202 controllers (RQHD1).

HAND4.CFG: Declaration of initial program counter for controller task code for iSBC 204 controllers (RQHD4).

The following files contain declarations of the system exchanges to which requests for DFS services are sent.

ATTRIB.EXT: Declaration of RQATRX.

DELETE.EXT: Declaration of RQDELX.

DIRECT.EXT: Declaration of RQDIRX.

DISKIO.EXT: Declaration of RQDSKX.

FORMAT.EXT: Declaration of RQFMTX.

LOAD.EXT: Declaration of RQLDX.

OPEN.EXT: Declaration of RQOPNX.

RENAME.EXT: Declaration of RQRNMX.

The following files contain definitions of various parameter values, declarations of general data structures, etc., that DFS users may find helpful. The most-used of these files are printed here for your convenience.

ATTRIB.ELT: Values for ATTRIB SWID parameter.

```
DECLARE
        ATR$INV            LITERALLY '0',
        ATR$SYS            LITERALLY '1',
        ATR$WTP            LITERALLY '2',
        ATR$FMT            LITERALLY '3';

DECLARE
        INVISIBLE$ATTRIBUTE    LITERALLY '01H',
        SYSTEM$ATTRIBUTE       LITERALLY '02H',
        WRITEP$ATTRIBUTE       LITERALLY '04H',
        FORMAT$ATTRIBUTE       LITERALLY '80H';
```

BAB.ELT: Declaration of a Buffer Allocation Block.

```
DECLARE BAB$ENTRY LITERALLY 'STRUCTURE (
        BUF$SOURCE         ADDRESS,
        BUF$RETURN         ADDRESS,
        NFILES             BYTE,
        BUF$PTR            ADDRESS)';
```

CST.ELT: Declaration of a Controller Specification Table entry.

```
DECLARE CST$ENTRY LITERALLY 'STRUCTURE (
        CONTROLLER$TYPE    BYTE,
        IO$BASE$ADR        BYTE,
        INTRP$LEVEL        BYTE,
        INTRP$XCH          ADDRESS,
        REQUEST$XCH        ADDRESS)';
```

DCT.ELT: Declaration of a Device Configuration Table entry.

```
DECLARE DCT$ENTRY LITERALLY 'STRUCTURE (
        DEVICENAME(2)      BYTE,
        DEVICETYPE         BYTE,
        CONTROLLER         BYTE,
        UNIT               BYTE)';
```

DEVCMD.ELT: Definition of IOINS values for DISKIO operations.

```
DECLARE SEEK$COMMAND       LITERALLY '1',
        FORMAT$COMMAND     LITERALLY '2',
        RECALIBRATE        LITERALLY '3',
        READ$COMMAND       LITERALLY '4',
        VERIFY$COMMAND     LITERALLY '5',
        WRITE$COMMAND      LITERALLY '6',
        WRITE$DEL          LITERALLY '7',
        MOTOR$ON           LITERALLY '8',
        MOTOR$OFF          LITERALLY '9',
        RESERVED           LITERALLY '10';
```

DFSMSG.ELT: Declaration of DFS request messages.

```
DECLARE ATTRIB$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS              ADDRESS,
        FILE$PTR            ADDRESS,
        SWID                ADDRESS,
        VALUE               ADDRESS)';

DECLARE CLOSE$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS              ADDRESS)';

DECLARE DELETE$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS              ADDRESS,
        FILE$PTR            ADDRESS)';

DECLARE DSK$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS              ADDRESS,
        DEVICE              BYTE,
        IOINS               ADDRESS,
        NSEC                BYTE,
        TADR                ADDRESS,
        SADR                ADDRESS,
        BUFFER              ADDRESS)';

DECLARE FORMAT$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS              ADDRESS,
        FILEPTR             ADDRESS)';

DECLARE LOAD$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS              ADDRESS,
        FILE$PTR            ADDRESS,
        BIAS                ADDRESS,
        ENTRY               ADDRESS)';

DECLARE OPEN$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS              ADDRESS,
        FILE$PTR            ADDRESS,
        ACCESS              ADDRESS,
        AFR$XCH             ADDRESS)';

DECLARE READ$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS              ADDRESS,
        BUFFER              ADDRESS,
        COUNT               ADDRESS,
        ACTUAL              ADDRESS)';

DECLARE RENAME$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS              ADDRESS,
        OLD$FILE$PTR        ADDRESS,
        NEW$FILE$PTR        ADDRESS)';
```

```
DECLARE SEEK$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS                  ADDRESS,
        MODE                    ADDRESS,
        BLOCKNO                 ADDRESS,
        BYTENO                  ADDRESS)';

DECLARE WRITE$REQ$MSG LITERALLY 'STRUCTURE (
        MSGHDR,
        STATUS                  ADDRESS,
        BUFFER                  ADDRESS,
        COUNT                   ADDRESS,
        ACTUAL                  ADDRESS)';
```

DFSREQ.ELT: Definition of DFS message types.

```
DECLARE
        DFS$RD                  LITERALLY '8',
        DFS$WRT                 LITERALLY '12',
        DFS$SK                  LITERALLY '13',
        DFS$CLS                 LITERALLY '14',
        DFS$OPN                 LITERALLY '15',
        DFS$RNM                 LITERALLY '16',
        DFS$DEL                 LITERALLY '17',
        DFS$ATR                 LITERALLY '18',
        DFS$LD                  LITERALLY '19',
        DFS$FMT                 LITERALLY '20',
        DFS$DSK                 LITERALLY '21';
```

FNB.ELT: Declaration of a File Name Block.

```
DECLARE FILE$NAME$BLOCK LITERALLY 'STRUCTURE (
        DEVICE$NAME(2)          BYTE,
        FILE$NAME(6)            BYTE,
        EXTENSION(3)            BYTE)';
```

OPNACC.ELT: Values for OPEN ACCESS mode parameter.

```
DECLARE RD$ACC                  LITERALLY '1',
        WRT$ACC                 LITERALLY '2',
        UPD$ACC                 LITERALLY '3';

DECLARE READ$ACCESS             LITERALLY '1',
        WRITE$ACCESS            LITERALLY '2',
        UPDATE$ACCESS           LITERALLY '3';    ·
```

SEEK.ELT: Values for SEEK MODE parameter.

```
DECLARE
        SK$CUR                  LITERALLY '0',
        SK$DECR                 LITERALLY '1',
        SK$SET                  LITERALLY '2',
        SK$INCR                 LITERALLY '3',
        SK$EOF                  LITERALLY '4';
```

The following files contain PL/M declarations commonly needed by users of the
Analog Handlers. The most-used of these files are printed here for your convenience.

AIHPRC.EXT: Declaration of the RQAIH procedure.

AOHPRC.EXT: Declaration of the RQAOH procedure.

AIH.EXT: Declaration of the RQAIEX exchange.

AOH.EXT: Declaration of the RQAOEX exchange.

AIMSG.ELT: Declaration of analog input request message.

```
DECLARE AIMSG LITERALLY 'STRUCTURE (
        MSG$HDR,
        STATUS              ADDRESS,
        BASE$PTR            ADDRESS,
        CHANNEL$GAIN        ADDRESS,
        ARRAY$PTR           ADDRESS,
        COUNT               ADDRESS,
        ACTUAL$COUNT        ADDRESS)';
```

AOMSG.ELT: Declaration of analog output request message.

```
DECLARE AIREP               LITERALLY '30',
        AISQS               LITERALLY '31',
        AISQV               LITERALLY '32',
        AIRAN               LITERALLY '33';
```

AITYP.ELT: Definition of analog input message types.

```
DECLARE AOMSG LITERALLY 'STRUCTURE (
        MSG$HDR,
        STATUS              ADDRESS,
        BASE$PTR            ADDRESS,
        ARRAY1$PTR          ADDRESS,
        ARRAY2$PTR          ADDRESS,
        COUNT               ADDRESS)';
```

AOTYP.ELT: Definition of analog output message types.

```
DECLARE AORAN               LITERALLY '38';
```


Assembly Language INCLUDE Files

The assembly language source code files listed in this section are useful when coding
an assembly language configuration module for a system that includes DFS services.
To include any of these sections of code in your progrom, you need not write out the
code — simply provide the name of the appropriate file in an assembler $INCLUDE
command.

RMXXCH.MAC: Code for the XCH, INTXCH, and PUBXCH
                              macros.

DFSCFG.MAC: Code for the CONSTD, CST, DCT, DRC4,
                              GENDRC, and BAB macros.

This appendix is a summary of the currently defined public and external symbols within the RMX/80 system. Public symbols are those that can be addressed from a user task. Each user task or module that addresses an RMX/80 public symbol must declare that symbol external. The user task or module that defines an RMX/80 external symbol must declare it public.

Publics Defined by RMX/80

Procedures

RQSEND:	Send Message operation.
RQWAIT:	Wait for Message operation.
RQACPT:	Accept Message operation.
RQCTCK:	Clock Tick operation (iSBC 80/10 version only).
RQCXCH:	Create Exchange operation.
RQCTSK:	Create Task operation.
RQDXCH:	Delete Exchange operation.
RQDTSK:	Delete Task operation.
RQSUSP:	Suspend Task operation.
RQRESM:	Resume Task operation.
RQDLVL:	Disable Level operation.
RQELVL:	Enable Level operation.
RQISND:	Interrupt Send operation.
RQENDI:	End Interrupt operation.
RQSETP:	Set Poll operation (iSBC 80/10 version only).
RQSETV:	Set Vector operation.

Data Structures and Variables

RQL1EX:	Level 1 interrupt exchange.
RQACTV:	Address of Task Descriptor of running task.
RQCTAB:	Terminal Handler control character table.
RQDBUG:	Terminal Handler debug input exchange.

RQINPX:	Terminal Handler input exchange.
RQOUTX:	Terminal Handler output exchange.
RQALRM:	Terminal Handler alarm output exchange.
RQWAKE:	Debugger wakeup exchange (declared in Terminal Handler).
RQFSAX:	Free Space Manager allocation exchange.
RQFSRX:	Free Space Manager reclamation exchange.
RQFREE:	Free Space Manager list of free (i.e., available) messages.
RQATRX:	Disk File System ATTRIB request exchange.
RQDELX:	Disk File System DELETE request exchange.
RQDIRX:	Disk File System directory services request exchange.
RQDSKX:	Disk File System DISKIO request exchange.
RQFMTX:	Disk File System FORMAT request exchange.
RQLDX:	Disk File System LOAD request exchange.
RQOPNX:	Disk File System OPEN request exchange.
RQRNMX:	Disk File System RENAME request exchange.
RQAIEX:	Analog Input Handler request exchange.
RQAOEX:	Analog Output Handler request exchange.
RQNAME:	Bootstrap Loader pointer to file name.
RQRSTV:	Bootstrap Loader RST interrupt vector array.
RQBOTX:	Bootstrap Loader token message exchange.
RQLODX:	Bootstrap Loader wakeup exchange.

Tasks

RQTHDI:	Terminal Handler input task.
RQTHDO:	Terminal Handler output task.
RQFMGR:	Free Space Manager task.
RQADBG:	Active Debugger task.
RQPDBG:	Passive Debugger task.
RQPATR:	Disk File System ATTRIB task.
RQPDEL:	Disk File System DELETE task.
RQPDIR:	Disk File System directory services task.

RQPDSK:	Disk File System DISKIO task.
RQHD1:	Disk File System controller handler task for iSBC 201 and 202 controllers.
RQHD4:	Disk File System controller handler task for iSBC 204 controllers.
RQPFMT:	Disk File System FORMAT task.
RQPLD:	Disk File System LOAD task.
RQPRNM:	Disk File System RENAME task.
RQHD1V:	Disk File System interrupt polling routine for iSBC 201 and 202 controllers (iSBC 80/10 version only).
RQHD4V:	Disk File System interrupt polling routine for iSBC 204 controllers (iSBC 80/10 version only).
RQAIH:	Analog Input Handler task.
RQAOH:	Analog Output Handler task.
RQBOOT:	Bootstrap Loader task.

Externals Referenced

The following identifiers are referenced by RMX/80 and should be supplied and declared public by the user.

RMX/80 provides the UNRSLV.LIB relocatable object code file to resolve external references not used in your system. UNRSLV.LIB provides default procedures for the externals RQINTI and RQCLKI (iSBC 80/10 version only), and default values for RQTCNT and RQRATE. The default for RQTCNT is 50. The default for RQRATE is zero; this value specifies the automatic baud rate search in the iSBC 80/20 and 80/30 versions. (If the baud rate search is not desired, or if you are using the iSBC 80/10 version on the minimal Terminal Handler, you must supply a value for RQRATE at configuration time.) Descriptions and listings for the default RQINTI and RQCLKI procedures are given in Appendix G.

BOTUNR.LIB takes the place of UNRSLV.LIB when linking a Bootstrap Loader System. In addition to the values supplied in UNRSLV.LIB, BOTUNR.LIB provides the default file name (RMXSYS) for the Bootstrap Loader as well as several internal values.

A similar library file, DFSUNR.LIB, is provided on the Extensions diskette to resolve external references made by the Disk File System. Default values are supplied by DFSUNR.LIB for RQMOTM (20 System Time Units, or one second on an iSBC 80/20), and RQTOV (200 System Time Units, or ten seconds on an iSBC 80/20).

In certain cases, the LINK program issues warning messages for unused interrupt exchanges. You may ignore these warnings as long as your system never refers to the interrupt exchanges or enables the corresponding levels with RQELVL.

Versions of the procedures RQINTI, RQSTRC, RQSTPC, and RQTPOL are supplied in SBC104.LIB for use with the iSBC 80/10 Demonstration System.

RQCRTB:	Create Table.

RQLAEX:	Level A interrupt exchange (iSBC 80/30 version only).
RQLBEX:	Level B interrupt exchange (iSBC 80/30 version only).
RQLCEX:	Level C interrupt exchange (iSBC 80/30 version only).
RQL0EX:	Level 0 interrupt exchange.
RQL2EX:	Level 2 interrupt exchange.
RQL3EX:	Level 3 interrupt exchange.
RQL4EX:	Level 4 interrupt exchange.
RQL5EX:	Level 5 interrupt exchange.
RQL6EX:	Level 6 interrupt exchange.
RQL7EX:	Level 7 interrupt exchange.
RQRATE:	Parameter indicating Terminal Handler baud rate.
RQCST:	Disk File System Controller Specification Table.
RQDCT:	Disk File System Device Configuration Table.
RQDRC4:	Disk File System Drive Characteristics Table for iSBC 204 controllers.
RQBAB:	Disk File System Buffer Allocation Block.
RQDBUF:	Disk File System internal buffer space (number in bytes).
RQNDEV:	Disk File System number of devices (drives).
RQMOTM:	Time delay (in System Time Units) before the Disk File System returns from the DISKIO MOTOR ON operation.
RQTOV:	Time delay (in System Time Units) before the Disk File System returns a drive timeout error code if the drive does not respond.
RQTCNT:	Number of clock ticks per System Time Unit (iSBC 80/10 version only).
RQINTI:	Interrupt Initialize operation (iSBC 80/10 version only).
RQCLKI:	Clock Initialize operation (iSBC 80/10 version only).
RQSTRC:	Start Clock operation (iSBC 80/10 version only).
RQSTPC:	Stop Clock operation (iSBC 80/10 version only).
RQTPOL:	Timer Poll operation (iSBC 80/10 version only; referenced only if RQINTI procedure, from SBC104.LIB, is used).
RQPOOL:	Buffer space for the Bootstrap Loader. The space referred to by this label can be used by other tasks when the Loader is inactive.
RQFILE:	Name of initial file to be loaded by Bootstrap Loader.

This appendix is a summary of the message types currently defined by RMX/80. The following codes appear in the TYPE field of the message heading — either provided there by a user task before sending the message to an RMX/80 extension task, or supplied by RMX/80 when it sends the message. For details on particular message types, refer to Chapter 2 and to the chapters on the various RMX/80 extensions.

The PL/M symbolic names listed below for the type codes are supplied in the IN-CLUDE files MSGTYP.ELT, DFSREQ.ELT, AITYP.ELT, and AOTYP.ELT. PL/M programmers should INCLUDE these files in their programs and use the symbolic names for messages types. Assembly language programmers must code the numeric values.

Type codes 0 through 63 are reserved for RMX/80 use, although not all of these codes are currently defined. Types 64 through 255 are available to be defined by the user.

TYPE CODE	DEFINITION
DATA$TYPE = 0:	Message containing data of no specific type.
INT$TYPE = 1:	Message generated by RMX/80 as the result of an interrupt at the level associated with the exchange from which the message was obtained. A message of this type is read-only and has a LENGTH of 5.
MISSEDINTTYPE = 2:	Message generated by RMX/80 when the system detects that an interrupt has been missed by the task(s) that service the interrupt level associated with the exchange. A message of this type is read-only and has a LENGTH of 5.
TIMEOUTTYPE = 3:	Message generated by RMX/80 when a task times out on a timed wait. A message of this type is read-only and has a LENGTH of 5.
FSREQTYPE = 4:	Free Space Manager conditional allocation request message.
UCREQTYPE = 5:	Free Space Manager unconditional allocation request message.
FSNAKTYPE = 6:	Free Space Manager negative acknowledge message.
CNTRLCTYPE = 7:	Message sent by the Terminal Handler to the RQWAKE exchange when the operator types an ASCII control-C on the keyboard to invoke the Debugger.
READ$TYPE = 8:	Terminal Handler read request message.
DFS$RD = 8:	Disk File System READ request message.
CLRRDTYPE = 9:	Terminal Handler clear-and-read request message.

TYPE CODE	DEFINITION
LASTRDTYPE = 10:	Terminal Handler last-read request message.
ALARM$TYPE = 11:	Terminal Handler alarm output request message.
WRITE$TYPE = 12:	Terminal Handler write request message.
DFS$WRT = 12:	Disk File System WRITE request message.
DFS$SK = 13:	Disk File System SEEK request message.
DFS$CLS = 14:	Disk File System CLOSE request message.
DFS$OPN = 15:	Disk File System OPEN request message.
DFS$RNM = 16:	Disk File System RENAME request message.
DFS$DEL = 17:	Disk File System DELETE request message.
DFS$ATR = 18:	Disk File System ATTRIB request message.
DFS$LD = 19:	Disk File System LOAD request message.
DFS$FMT = 20:	Disk File System FORMAT request message.
DFS$DSK = 21:	Disk File System DISKIO request message.
AIREP = 30:	Analog Input Handler request message for repetitive single-channel input.
AISQS = 31:	Analog Input Handler request message for sequential channel input with a single gain.
AISQV = 32:	Analog Input Handler request message for sequential channel input with variable gain.
AIRAN = 33:	Analog Input Handler request message for random channel input.
AORAN = 38:	Analog Output Handler request message for random channel output.

Memory Requirements for RMX/80 Modules and Tables

This section of the appendix lists the memory requirements for the modules included in Version 1.3 of RMX/80, and the additional tables required for configuration. Stack requirements are included in the RAM byte counts. The RAM counts for the Terminal Handler, Free Space Manager, and Debugger modules also include table allocations required for internal use.

If your iSBC installation is equipped with a disk and you choose to use the Bootstrap Loader extension, many of the modules listed as having ROM requirements can be located entirely in RAM. In fact, all modules except the Nucleus, DISKIO and those contained in the Loader System Library (BOT8xO.LIB) can be moved completely out of ROM. As a reminder of this fact, only those modules marked with an asterisk must remain in ROM.

Modules on Executive Diskette (iSBC 80/20 Version)

Module	ROM (bytes)	RAM (bytes)
80/20 Nucleus*	1799	224
Optional Nucleus capabilities:		
RQDTSK	73	0
RQDXCH	31	0
RQSUSP	59	0
RQRESM	46	0
Terminal Handler, full (input-output version)	3088	941
Terminal Handler, full (output-only version)	935	219
Terminal Handler, minimal (input-output version)	599	156
Terminal Handler, minimal (output-only version)	167	27
Free Space Manager	1025	260
Active Debugger	11363	1706
Passive Debugger	4007	636

Modules on Executive Diskette (iSBC 80/30 Version)

Module	ROM (bytes)	RAM (bytes)
80/30 Nucleus*	1977	242
Optional Nucleus capabilities:		
RQDTSK	73	0
RQDXCH	31	0
RQSUSP	59	0
RQRESM	46	0
Terminal Handler, full (input-output version)	3088	941
Terminal Handler, full (output-only version)	935	219
Terminal Handler, minimal (input-output version)	599	156
Terminal Handler, minimal (output-only version)	167	27
Free Space Manager	1025	260
Active Debugger	11363	1706
Passive Debugger	4007	636

* Those modules marked with an asterisk must remain in ROM even in systems incorporating the Bootstrap Loader extension. All other modules used in such a system can be located entirely in RAM.

Modules on Executive Diskette (iSBC 80/10 Version)

Module	ROM (bytes)	RAM (bytes)
80/10 Nucleus*	1786	225
Optional Nucleus capabilities:		
RQDTSK	73	0
RQDXCH	35	0
RQSUSP	67	0
RQRESM	54	0
Terminal Handler, full (input-output version)	3268	966
Terminal Handler, full (output-only version)	1194	243
Terminal Handler, minimal (input-output version)	669	156
Terminal Handler, minimal (output-only version)	207	27
Free Space Manager	1025	261
Active Debugger	11748	1706
Passive Debugger	4082	636

Modules on Extensions Diskette

Several points need to be noted about the byte counts listed for Disk File System modules. First, the numbers in parentheses in the RAM column indicate the portion of RAM that must be controller-addressable. Second, the total memory space required for files is the amount specified for each open file times the maximum number of files that are ever open concurrently. Third, the LOAD service opens one file; so when LOAD is included in your system, you must take this fact into account in determining the maximum number of concurrently open files.

Finally, when FORMAT is included in your system, you must also add in the byte count(s) for one or more of the following dynamically created tasks, whichever one(s) are applicable to your system: FORMAT201 (standard-size single-density diskettes), FORMAT202 (standard-size double-density diskettes), and/or FORMAT-204 (mini-size diskettes).

Module	ROM (bytes)	RAM (bytes)
Disk File System:		
DISKIO*	255	13
DIRSVC	4960	1220 (700)
SEEK (iSBC 80/20 and 80/30 versions)	925	0
SEEK (iSBC 80/10 version)	957	0
LOAD	615	52
FORMAT	1962	51
FORMAT201	89	0
FORMAT202	93	0
FORMAT204	91	0
ATTRIB	180	99
RENAM E	264	40
DELETE	86	33
RQHD1 (iSBC 80/20 and 80/30 versions)*	455	0
RQHD1 (iSBC 80/10 version)*	497	20
RQHD4 (iSBC 80/20 and 80/30 versions)	752	0
RQHD4 (iSBC 80/10 version)	795	20
Each controller task	20	80 (80)
Each open file	0	400 (400)
Analog Input Handler	594	21
Analog Output Handler	180	19

* Those modules marked with an asterisk must remain in ROM even in systems incorporating the Bootstrap Loader extension. All other modules used in such a system can be located entirely in RAM.

Bootstrap Loader:
    Loader System Library

BOT810.LIB*	589	384 (256)
BOT820.LIB*	557	384 (256)
BOT830.LIB*	557	384 (256)
Loaded System Library		
LOD810.LIB	0	216
LOD820.LIB	0	344
LOD830.LIB	0	439

Configuration Data (All Versions)

Table	ROM (bytes)	RAM (bytes)
Task Descriptor, per task	0	20
Exchange Descriptor, per exchange	0	10
Interrupt Exchange Descriptor, per interrupt exchange	0	15
Create Table	6	0
Initial Task Table, per entry	17	0
Initial Exchange Table, per entry	2	0
DFS Controller Specification Table, per entry	7	0
DSF Device Configuration Table, per entry	5	0
DFS Drive Characteristics Table (iSBC 204), number of entries	1	0
DFS Drive Characteristics Table (iSBC 204), per entry	5	0
DFS Buffer Allocation Block	5	0
RQNDEV (number of disk drives in system)	1	0

Stack Size Requirements for RMX/80 Operations

The RMX/80 Nucleus operations use the stacks of the tasks that call them.
Therefore, in order to estimate the maximum stack length required for your applica-
tion tasks, you must know the stack requirements of the Nucleus operations. These
are listed below. Note that these numbers do not include the two bytes required by
the CALL instruction used to call the operation; these two bytes are part of the stack
requirement of the user code and should be included in the stack requirement given
by the compiler or assembler. (Refer to "Estimating Stack Length" in Chapter 3.)

The RQEND1, RQISND, and RQCTCK operations are not included in this list,
because these operations should not be called from user tasks. RQEND1 and
RQISND are only to be called from user-supplied interrupt service routines;
RQCTCK (iSBC 80/10 version only) may be called only from an interrupt polling
routine. The 24 extra bytes in the formula in Chapter 3 take into account the stack
requirement for these operations.

Operation	Stack Requirement (bytes)
RQACPT	4
RQCTSK	8
RQCXCH	0
RQDLVL	0
RQDTSK	2

* Those modules marked with an asterisk must remain in ROM even in systems
  incorporating the Bootstrap Loader extension. All other modules used in such a
  system can be located entirely in RAM.

RQDXCH	2
RQELVL	0
RQRESM	2
RQSEND	0
RQSETP	0
RQSETV	0
RQSUSP	2
RQWAIT	4

Memory Requirements for Four Sample Configurations

The following examples illustrate how you can calculate the total ROM and RAM requirements for your application systems. Memory requirements for the RMX/80 Nucleus and extension tasks are obtained from the first part of this appendix; you must determine the memory requirements for your own tasks. (The latter must include stack requirements; refer to Chapter 3 and to the second section of this appendix.)

Example 1

The following chart lists the memory requirements for a simple RMX/80 system. The system has four user tasks, each of which requires 700 bytes for code and 100 bytes of data storage. The system includes the RMX/80 Nucleus and Terminal Handler. Notice that this system can be fully contained on an iSBC 80/20.

Module or Table	ROM (bytes)	RAM (bytes)
80/20 Nucleus	1799	244
80/20 Terminal Handler (input-output version)	3088	941
User Tasks	2800	400
User Task Descriptors	0	80
User Exchange Descriptors (6)	0	60
User Interrupt Exchange Descriptors (2)	0	30
Initial Task Table	68	0
Initial Exchange Table	16	0
Create Table	6	0
	7777	1735

Example 2

The next example is for an iSBC 80/20 application system which uses the Disk File System. The DFS services used are OPEN, READ, WRITE, SEEK and CLOSE. Dynamic buffer allocation is to be used, so the Free Space Manager is included. There are two iSBC 201 disk controllers, each with its own task; one controller has two drives, the other has one. The maximum number of files open at the same time is two. There are two user tasks which require a total of 1250 bytes for code and 600 bytes for data storage; 320 bytes of the data storage is controller-addressable (for controller task stacks and one user buffer).

Parenthesized numbers in the RAM column indicate the portion of RAM which must be controller-addressable.

Module	ROM (bytes)	RAM (bytes)	
80/20 Nucleus (including RQDTSK, and RQDXCH, which are required by the Disk File System)	1903	224	
Free Space Manger	1025	260	
DISKIO	255	13	
DIRSVC	4960	1220	(700)
SEEK	925	0	
RQHD1	455	0	
Controller tasks	40	160	(160)
Buffer space (maximum of 2 files open concurrently)	0	800	(800)
User tasks	1250	600	(320)
User Task Descriptors	0	120	
User Exchange Descriptors (4)	0	40	
User Interrupt Exchange Descriptors (2)	0	30	
Interrupt Exchange Descriptors	0	30	
Initial Task Table	102	0	
Initial Exchange Table	12	0	
Create Table	6	0	
Controller Specification Table	14	0	
Device Configuration Table	15	0	
Buffer Allocation Block	5	0	
RQNDEV (number of disk drives)	1	0	
	10968	3467	(1980)

Example 3

The next example shows how a minimal DFS directory-based system, using one iSBC 201 controller and two drives, can be implemented in the iSBC 80/20 on-board ROM. 492 bytes are left for a user-supplied task.

Parenthesized numbers in the RAM column indicate the portion of RAM which must be controller-addressable. Asterisks indicate that the amount of RAM needed for buffers depends upon the application.

Module	ROM (bytes)	RAM (bytes)	
80/20 Nucleus (including RQDTSK, and RQDXCH, which are required by the Disk File System)	1903	224	
DISKIO	255	13	
RQHD1	455	0	
DIRSVC	4960	1220	(700)
Controller tasks	20	80	(80)
Buffer space	0	*	(*)
User tasks	492	0	
User Task Descriptors	0	80	
User Exchange Descriptors (4)	0	40	
User Interrupt Exchange Descriptors	0	15	
Interrupt Exchange Descriptors	0	15	
Initial Exchange Table	10	0	
Initial Task Table	68	0	
Create Table	6	0	
Controller Specification Table	7	0	
RQNDEV (number of disk drives)	1	0	
Device Configuration Table	10	0	
Buffer Allocation Block	5	0	
	8192	1672*	(780)*

Example 4

The next example is for a Bootstrap Loader System implemented on an iSBC 80/20 using a Model 201 disk controller board. This is the smallest possible Loader System as it does not include any applications tasks. Should you wish to add applications tasks to the system, you also will have to consider their memory requirements.

In the table below, those numbers enclosed in parentheses indicate the portion of RAM which must be controller addressable. Also, the segment of RAM allocated to RQPOOL can be used by other tasks when the Loader Task is inactive.

Module	ROM (bytes)	RAM (bytes)	
80/20 Nucleus	1799	224	
Bootstrap Loader	557	128	
DISKIO	255	13	
RQHD1	455	0	
Controller Task Descriptor and Stack	20	80	(80)
RQPOOL	0	256	(256)
Configuration Module	90	200	
	3176	901	(336)

Since this system contains no tasks other than the Loader, it can load 20K bytes in less than 40 seconds. If additional tasks are included in the Loader System, this performance may be degraded.

This appendix describes the file structure that is used by ISIS-II and the RMX/80 Disk File System. The information is provided for those who want to know the details of how files are physically represented on disk. This information is not required to use ISIS-II or the RMX/80 Disk File System successfully.

Information is included on file structure for standard-size single-density diskettes, standard-size double-density diskettes, and mini-size diskettes. Note, however, that only the standard-size diskettes are ISIS-II compatible.

General File Structure

All ISIS-II and DFS files are made up of the same components, shown in figure E-1.
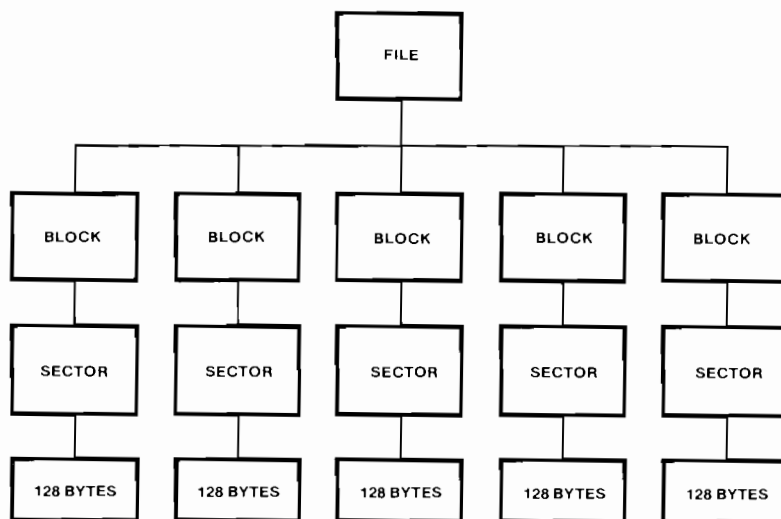
Figure E-1. Disk File Components

Each block corresponds to one disk sector, which is a hardware-addressable unit, 128 bytes long. Gross disk capacities are shown below:

	Standard-Size Single-Density	Standard-Size Double-Density	Mini-Size Single-Density
Tracks/Disk	77	77	35
Sectors/Track	26	52	18
Sectors/Disk	2,002	4,004	630
Bytes/Disk	256,256	512,512	80,640

Each sector on a disk has a unique address by which it can be accessed. The address consists of a one-byte track number and a one-byte sector (block) number. Tracks are numbered 0-76 on standard-size and 0-34 on mini-size disks; the sectors (blocks) on a track are numbered 1-26 on standard-size single-density disks, 1-52 on standard-size double-density disks, and 1-18 on mini-size disks. The address of a block is also referred to as a "pointer" to that block. Related blocks are linked together by these pointers (that is, two of the bytes in a block may contain the address of another block).

There are two types of blocks in a file: *pointer blocks* and *data blocks*. Pointer blocks contain nothing but pointers to other blocks, as shown in figure E-2.
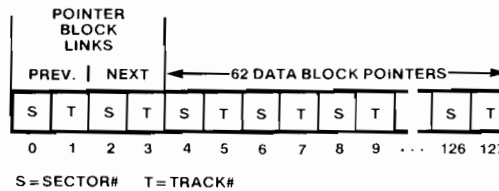
Figure E-2. Pointer Block

All files begin with a pointer block that is called the *header block* . If the file contains 62 or fewer data blocks, the header block is the only pointer block in the file. If there are more than 62 data blocks in the file, there is an additional pointer block for every 62 data blocks. A file of 300 data blocks contains 5 pointer blocks, including the header block.

The first two pointers in a pointer block are links to other pointer blocks in the file. The first link contains the address of the previous pointer block. The header block always contains zeros in this field because it is the address of the first block in the file. The second link contains the address of the next pointer block in the file; the last pointer block in the file has zeros in this link.

Following the pointer block links are 62 pointers to data blocks in the file. If a pointer contains zeros, then no data block has been allocated for the pointer. A zero pointer does not, however, necessarily mark the end of the file. Data blocks, as shown in figure E-3 have no particular format, since they contain user, as opposed to system, data.
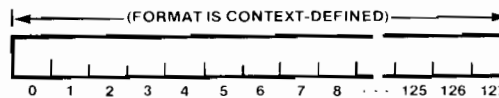
Figure E-3. Data Block

Data blocks are fundamentally different from pointer blocks. Data blocks are "visible" to users; they contain the information which is transferred by read and write operations. Pointer blocks, on the other hand, are "invisible" to users; the data they contain is of interest only to the system. Where data blocks are "destinations", pointer blocks are "paths" to those destinations. To access user data in a file, ISIS-II and DFS follow a path of pointers to a data block.

The relationship of pointer and data blocks in a file is illustrated in figure E-4.
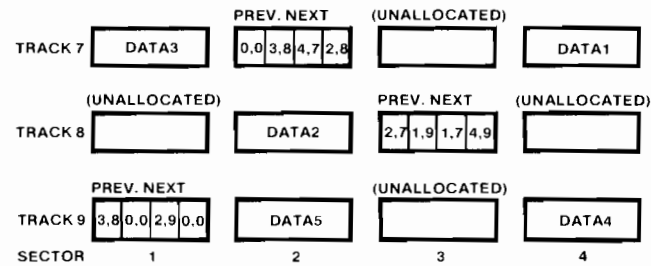
Figure E-4. Pointer and Data Blocks in a Typical File

The diagram is simplified in that it shows only four sectors per track (instead of 26, 52, or 18) and only two data pointers per pointer block (instead of 62). The file "begins" at the header block which contains pointers to the first two data blocks (DATA1 and DATA2). The header block is chained (linked) to a second pointer block located at sector 3 of track 8. This block contains pointers to DATA3 and DATA4, is chained back to the header block and forward to the last pointer block located at sector 1 of track 9. This block contains the last data pointer in the file. Because it is the last pointer block, it has a backward link but no forward link. Notice that it is the data pointers which order the data blocks for sequential access. The physical location of the data blocks (and the pointer blocks, for that matter) is immaterial. This ability to scatter files all over the disk allows the system to make very efficient use of available space. Note also that the effective capacity of a disk to store data blocks is diminished by the number of pointer blocks on the disk.

System Files

All ISIS-II and DFS non-system diskettes contain four system files. These are created automatically when the disk is formatted, and may be subsequently updated by the system. The location of these files is fixed as shown in table E-1.

Table E-1. System File Map

File Name		Standard-Size Single-Density		Standard-Size Double-Density		Mini-Size Single-Density	
		From	Thru	From	Thru	From	Thru
ISIS.TO	(Header)[1]	0,24[2]	0,24	0,24	0,24	0,16	0,16
	(Data)	0,1	0,23	0,1	0,23	0,16	0,15
ISIS.LAB	(Header)	0,25	0,25	0,25	0,25	0,17	0,17
	(Data)	0,26	0,26	0,26	0,52 &	0,18	0,18
				1,27	1,52		
ISIS.DIR	(Header)	1,1	1,1	1,1	1,1	1,1	1,1
	(Data)	1,2	1,26	1,2	1,26	1,2	0,18
ISIS.MAP	(Header)	2,1	2,1	2,1	2,1	2,1	2,1
	(Data)	2,2	2,3	2,2	2,5	2,2	2,2

1.   The header block is the only pointer block in each system file.

2.   Track 0, Sector 24.

ISIS.T0

This file contains a program called T0BOOT. When the BOOT and RESET buttons on the Intellec system (RESET button only on Intellec Series II) are pressed, this program is read in from the disk. There are two versions of T0BOOT: one for the ISIS-II or DFS system disks and one for all other disks. Control is passed to T0BOOT, which then reads in the rest of the system from the disk and displays "ISIS" and the ISIS-II version number on the console. If an attempt is made to boot the system from a non-system disk, the alternate version of T0BOOT displays "NON-SYSTEM DISKETTE - TRY ANOTHER" and gives control to the Monitor.

ISIS.LAB

The first 9 bytes of this file contain the name (label) of the disk stored in ASCII characters as follows: xxxxxx.yyy (name.extension). The next 40 characters are ASCII nulls (binary zeros). The next two bytes are a carriage return and line feed (0DH and 0AH). The next 77 bytes (35 bytes on a mini-size disk) contain a one-byte interleaving factor for each track on the disk as given in table E-2.

Table E-2. Sector Interleaving Factors

Disk Size and Density	Track 0	Track 1	Tracks 2-76
Standard-Size Single-Density	1	12	6
Standard-Size Double-Density	1	6	5
Mini-Size Single-Density	1	6	6

The content of the remainder of the file (double-density disks only) is undefined.

Interleaving factors are used to speed up the sequential access of blocks on the same track. Often a program reads a block, does a little processing, reads the next block, and so on. If the blocks were stored in physically adjacent sectors, the second sector would be passing the head while the program was processing the first sector. The program would have to wait nearly one full revolution for the second sector to come around again. The effect of an interleaving factor of 3 is shown in figure E-5.
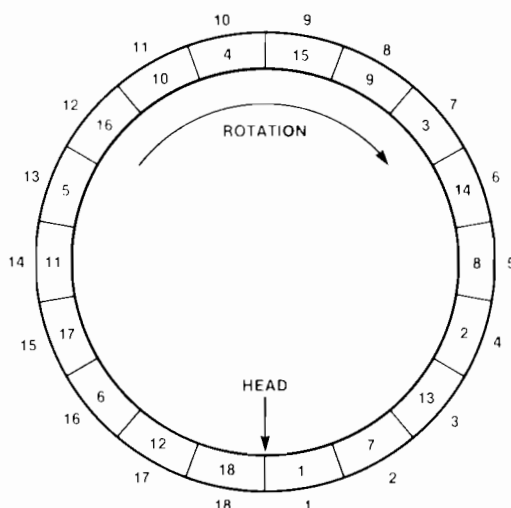
Figure E-5. Sector Interleaving

Physical sector addresses are shown outside the "track" (which is simplified to depict only 18 sectors). Logical sector addresses, which are the basis for accessing the blocks stored in the sectors, are shown on the "track." With interleaving, a program which reads and processes every block in logical sequence has a processing "window" equivalent to the time it takes for two sectors to pass the head. Assuming that processing each block takes slightly less time than is available in the window, all 18 blocks can be processed in three revolutions of the disk. Without interleaving, 18 revolutions would be required.

ISIS.DIR

This file contains 25 data blocks (17 for mini-size disks); each of these blocks has room for 8 directory entries. One entry is used for each file on the disk, so there is room in the directory for 200 files (136 for mini-size disks). Each directory entry is 16 bytes long and is formatted as shown in figure E-6.
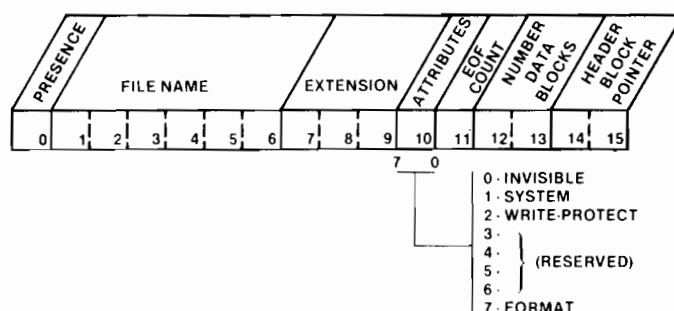
Figure E-6. Directory Entry

PRESENCE is a flag which can contain one of three values:

000H: The file associated with this entry is present on the disk.

07FH: No file is associated with this entry; the content of the rest of the entry is undefined. The first entry with its flag set to 07FH marks the current logical end of the directory, and directory searches stop at this entry.

0FFH: The file named in this entry once existed on the disk, but is currently deleted. The next file added to the directory will be placed in the first entry marked 0FFH. This flag cannot, therefore, be used to (reliably) find a file which has been deleted. A value of 0FFH should be thought of as simply marking an open directory entry.

FILE NAME is a string of up to 6 non-blank ASCII characters specifying the name of the file associated with the directory entry. If the file name is shorter than six characters, the remaining bytes contain binary zero. For example, the name ALPHA would be stored as 414C50484100H.

EXTENSION is a string of up to 3 non-blank ASCII characters that specify an extension to the file name. Extensions often identify the type of data in the file such as OBJ (object module), or PLM (PL/M source module). As with the file name, unused positions in the extension field are filled with binary zeros.

ATTRIBUTES are bits that identify certain characteristics of the file. A 1 bit indicates that the file has the attribute, while a 0 bit means that the file does not have

the attribute. The bit positions and their corresponding attributes are listed below (bit 0 is the low-order or rightmost bit, bit 7 is the leftmost bit):

0:            Invisible. Files with this attribute are not listed by the ISIS-II DIR command unless the I switch is used. All system files are invisible.

1:            System. Files with this attribute are copied to the disk in drive 1 when the S switch is specified with the ISIS-II FORMAT command.

2:            Write-Protect. Files with this attribute cannot be opened for output or update, nor can they be deleted or renamed.

3-6:   These positions are reserved for future use.

7:            Format. Files with this attribute are treated as though they are write-protected. In addition, these files are created on a new diskette when the disk is formatted via the ISIS-II FORMAT command; however, the DFS FORMAT service does not do this. The system files all have the FORMAT attribute, and this attribute should not be given to any other files.

The ISIS-II System User's Guide, 9800306, should be consulted for more information on attributes.

EOF COUNT contains the number of the last byte in the last data block of the file. If the value of this field is 080H, for example, the last byte in the file is byte number 128 in the last data block (the last block is full).

NUMBER OF DATA BLOCKS is an address variable which indicates the number of data blocks currently used by the file. ISIS-II and the RMX/80 Disk File System both maintain a counter called LENGTH, which contains the current number of bytes in the file. This is calculated as:

   ((NUMBER OF DATA BLOCKS) x 128) + EOF COUNT.

HEADER BLOCK POINTER is the address of the file's header block. The high-order byte of this field is the sector number, and the low-order byte is the track number. The system "finds" a disk file by searching the directory for the name, then using the header block pointer to seek the beginning of the file.

ISIS.MAP

This file contains a bit map of the disk, with each bit position representing one block (sector) or the disk. If a bit is 1, the corresponding block is allocated, that is, in use as a pointer block or a data block. 0 bits denote free space on the disk. When a file is deleted, the bits that correspond to the sectors it occupied are reset to 0. On standard-size single-density disks, system files occupy the first two tracks and the first three sectors of the third track (in other words, the first 55 sectors). Therefore the first 55 bits of a standard-size single-density map are always 1. The last 46 map bits are always 0, since the map has 2048 bit posistions and there are only 2002 sectors on a single-density disk. For double-density disks the first 109 bits are always 1 and the last 92 bits are always 0. For mini-size disks the first 38 bits are always 1 and the last 376 are always 0.

File Structure Summary

Figure E-7 provides an overall view of the most important elements in the file structure. Some simplifications have been made for clarity (there are only four directory blocks, pointer blocks contain only four data block addresses, etc.), but the key relationships of file elements are shown.
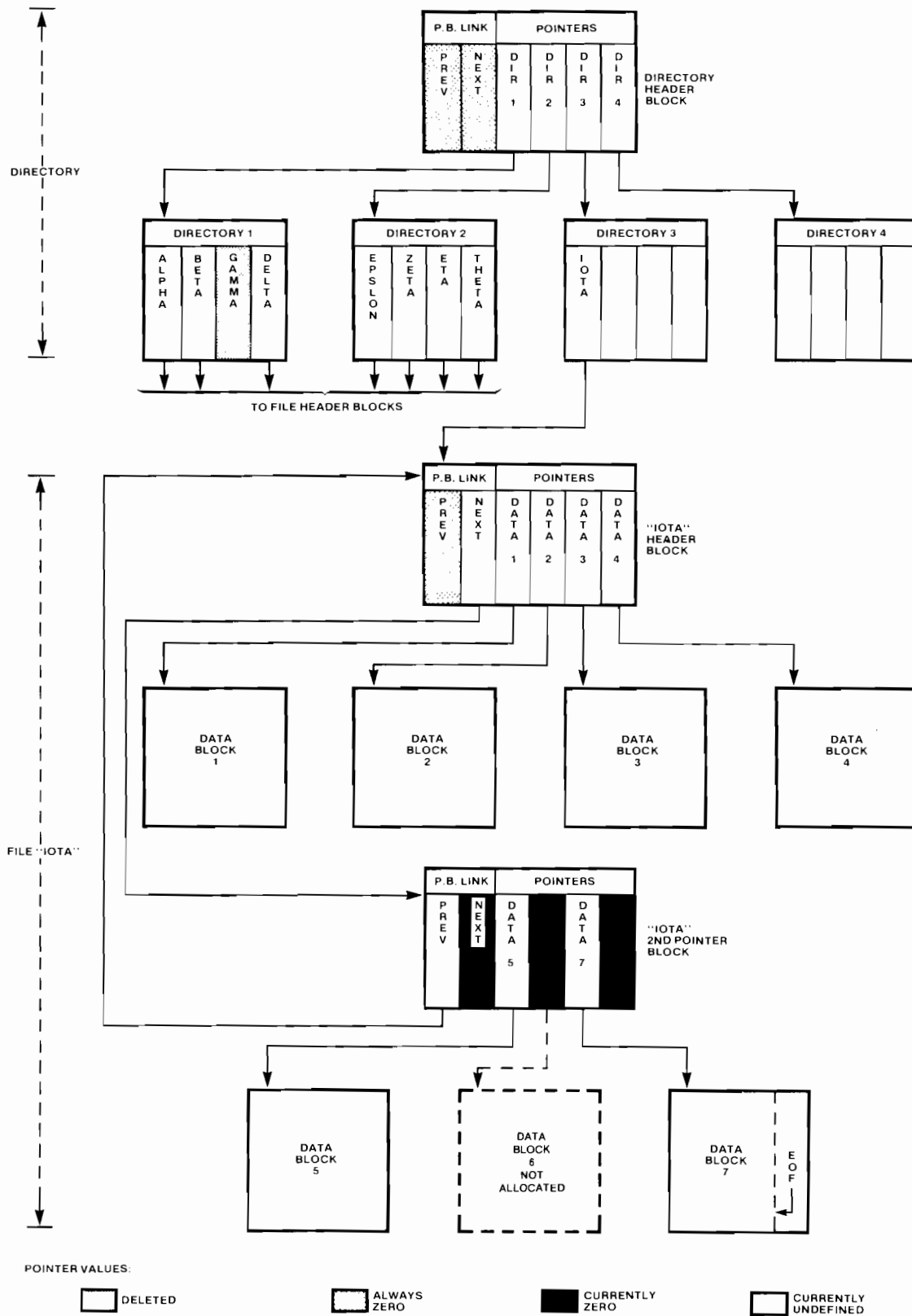
Figure E-7. Disk File Structure Summary

One new concept is introduced in the drawing, Notice that IOTA's second pointer block contains a zero pointer between the pointer to data block 5 and the pointer to data block 7. Data block 6 does not exist (no space is allocated to it), but it can be added later, since there is a "slot" left open in the pointer block. If an attempt is made to read data block 6, the operation will be performed normally, but binary zeros will be placed in the requesting program's buffer. If an attempt is made to write data block 6, a new data block will be allocated and written and its address will be placed in the pointer block. Blocks such as data block 6, which are "logically allocated" but not physically allocated, are included in the NUMBER OF DATA BLOCKS field of the directory.

IOTA is an example of how a "sparse" file can be implemented efficiently within the file structure. Such a file is created by opening the file for update and writing records where they occur and seeking ahead when there is no record. For example, assume IOTA is a file which can ultimately contain seven 128-byte records (ignore the EOF mark in the drawing). However, when the file is created there is no data for record six. The file would be created by opening it for update, writing the first five records in sequence, seeking ahead 128 bytes, and writing record seven.

This appendix provides hardware installation procedures that must be performed in order to run your RMX/80-based software on an iSBC 80/20, 80/30, or 80/10 system. These procedures apply in addition to the installation procedures prescribed in the hardware manuals for your equipment.

Each of the three iSBC 80 boards is discussed in a separate section.

iSBC 80/20

Terminal Interface

The standard Intel terminal is cabled to communicate with an Intel Microcomputer Development System rather than a Single Board Computer. Therefore, an Intel terminal with an unmodified cable will not work with an iSBC 80/20 (or 80/30 or 80/10).

When using the serial I/O port, the iSBC 80 requires the Request-to-Send (RTS) signal at pin 8 of J3. This corresponds to pin 4 of the 25-pin serial I/O connector commonly in use. Adding pin 4 to the 25-pin connector CRT cable allows the CRT to function properly. When using a different terminal that does not generate the RTS signal, a jumper can be added to the iSBC 80 between pins 8 and 10 of J3, or between pins 4 and 5 of the 25-pin connector. This routes the iSBC-generated CTS (Clear-to-Send) signal back into the board as an RTS signal.

iSBC 80/20 Interrupts

The RMX/80 Terminal Handler uses interrupts 6 and 7 for the READ and WRITE interrupts, respectively. The RXRDY and TXRDY signals from the USART (8251 Programmable Communications Interface) are used to generate these interrupts. The time base used by RMX/80 is generated from the 8253 Programmable Interval Timer chip.

To accomplish this interrupt configuration, the following wiring changes must be made:

Remove	25 - 45	(1R1 - INT2 on P1)
	35 - 26	(OIT0 - IR2)
Connect	31 - 39 ⎫	
	39 - 38 ⎬	Ground multiple inputs to IR7
	38 - 37 ⎭	
	35 - 25	OIT0 - IR1
	41 - 30	RXR - IR6
	40 - 36	TXR - IR7

Bus Priority

To use the iSBC 80/20 (or 80/30) as the bus master, you must connect its BPRN pin so as to grant it bus access when it needs to use the bus. This can be done by grounding the BPRN pin or by tying it to the BREQ pin to let it grant itself bus access when desired.

The iSBC 80 board can be plugged into any of the four slots, J2 through J5, of the 604 card cage. It comes installed in J5 with a jumper connecting pins G - H on the card cage mother board. In most systems, you will probably wish to place the iSBC 80/20 in slot J2 (top of cage) to allow the ICE-80 cable to plug in and still use the other slots for memory or other controllers. This requires an additional jumper to be installed. The table below shows the jumpers needed for each slot if the only bus master is the processor board.

SLOT	BPRN	BREQ
J2	B	A
J3	C	D
J4	E	F
J5	H	G

If other controllers are used, a daisy chain priority scheme must be implemented, as described in the iSBC 80/20 or 80/30 Hardware Reference Manual.

ICE-80 Processor Module Jumpers

The ICE-80 processor module must have its jumpers set for external clock and high frequency operation.

External Clock	A - C
	E - F
High Frequency	K - L
	J - H

ICE-80 8259 Fix

A small circuit board, available from Intel Customer Service, must be soldered onto chip A27 of the ICE-80 processor board to make it compatible with the iSBC 80/20. It is labeled PWA 1001359. If your system was shipped after September of 1977, then it already has this patch board. Otherwise, you should check visually to determine whether it has been incorporated into your system. Its purpose is to allow for the iSBC 80/20's use of 3-byte calls for interrupts instead of the RST instruction.

ICE-80 Mode Selection Switch

A special switch must be installed on the System 80/20 to handle the acknowledge signals generated for I/O bus and memory acknowledges. The switch is a 3-position single-pole, double-throw, center-off switch whose functions are labeled LOAD, OFF and RUN.

The LOAD position allows ICE-80 to load code into the iSBC 80/20 which has memory mapped physically into the Intellec development system. The ICE-80 module writes out the memory address and data to both the iSBC and development systems and must get an acknowledge from both systems. Otherwise, ICE-80 reports an error 67 (250 msec time out), which aborts the load process. When loading code into mapped memory, it is common to have no equivalent memory in the iSBC or for that memory to be ROM. In either case, this gives no acknowledge. A pseudo-acknowledge can be created by grounding pin 137, which gives an immediate acknowledge of the iSBC memory access.

The OFF position of the switch is useful for normal operation of the iSBC 80/20 without ICE-80 and with its own full complement of ROM and RAM.

The RUN position of the switch allows ICE-80 to get off-board acknowledges for all memory accesses to nonexistent memory; however, it will not acknowledge a write

to the on-board ROM, thus causing errors. These errors can be prevented with the following modifications:

> Disconnect 10 ms time out   137-138
>
> Connect Switch
>   terminal   1 - 135 (run position)
>   common   53 (ground)
>   terminal   2 - 137 (load position)

Accessing Memory in an iSBC 80 System Using ICE-80

There are two ways a user may wish to allocate memory in a system using both ICE-80 and a System 80/20 (or 80/10). The first is to use only Intellec Development System memory and map it into the iSBC 80 logical memory space. The use of the LOAD/RUN switch for this technique was described previously. This technique has the limitation of providing insufficient memory for large programs with correspondingly large symbol tables, since 12K is used for ISIS-II, 12 K for ICE-80, and 2K for the Monitor. This leaves 38K of Intellec Development System memory to be split between the symbol table and the mapped code.

To obtain more usable memory, 16K or 32K RAM boards may be plugged into the System 80/20 card cage. Up to four boards may be added when a 614 expansion chassis is used. The on-board PROM and RAM always take precedence over the off-board memory in the system. To bypass the on-board PROM, block 0 (0-4K) of memory must be mapped from the Development System using the command:

XFORM MEMORY 0 INTO 6

The 2K* of on-board RAM are used instead of the equivalent addresses of the 16K RAM board. A memory map would look like:

0- 4K	DEVELOPMENT SYSTEM MAPPED MEMORY
4-14K**	1st 16K RAM BOARD
14-16K***	ON BOARD MAP
16-32K	2nd 16K RAM BOARD
32-48K	3rd 16K RAM BOARD
48-64K	4th 16K RAM BOARD

---

  * 4K for the iSBC 80/20-4; 1K for the iSBC 80/10.

 ** 4-12K for the iSBC 80/20-4; 4-15K for the iSBC 80/10.

*** 12-16K for the iSBC 80/20-4; 15-16K for the iSBC 80/10.

To use memory in this way requires the LOAD/RUN switch to function differently: the LOAD setting must generate an acknowledge for stores into PROM. The acknowledge cannot be immediate as with Development System memory. An immediate acknowledge when storing into iSBC RAM is premature. Instead, wait for the normally generated RAM board acknowledge. The 10-ms timer on the iSBC 80/20 can be used to generate the PROM write acknowledge by connecting pins 137 and 138 via the switch's LOAD setting.

The RUN position needs no connections, since all acknowledges are properly generated by the standard hardware. The 10-ms timeout should not be connected, so that inadvertent stores to PROM may be caught by ICE-80 error 67 (timeout). Note that the two modes of using memory are mutually exclusive. Additional Develop-

ment System memory blocks cannot be mapped to logical areas of System 80 memory that do not already have physical memory in place. The only reason the first 4K may use Development System mapped memory is that the iSBC 80 generates acknowledges for reads to these addresses. If no RAM board is in place for iSBC memory space 48K to 64K, Development System memory mapped here will not get access acknowledged, and ICE-80 will generate error 67 timeouts.

iSBC 80/30

Terminal Interface

Refer to "Terminal Interface" in the iSBC 80/20 section of this appendix.

iSBC 80/30 Interrupts

The RMX/80 Terminal Handler uses interrupts 6 and 7 for the READ and WRITE interrupts, respectively. The RXRDY and TXRDY signals from the USART (8251 Programmable Communications Interface) are used to generate these interrupts. The time base used by RMX/80 is generated from the 8253 Programmable Interval Timer chip.

To accomplish this interrupt configuration, the following wiring changes must be made:

Remove	123 - 138	(COUNT OUT - INTR 7.5)
	46 - 47	(CLK1 - CLK0)
	47 - 52	(CLK0 - CLK2)
Connect	141 - 132	(EVENT CLK - IR1)
	47 - 51	(CLK0 - A12-11)
	143 - 127	(RXR INTR - IR6)
	142 - 126	(TXR INTR - IR7)
	145 - 140	(Ground - INTR 5.5)
	145 - 139	(Ground - INTR 6.5)

Bus Priority

Refer to "Bus Priority" in the iSBC 80/20 section of this appendix.

Accessing Memory in an iSBC 80/30 System Using ICE-85

ICE-85 provides two modes of memory access, USER and INTELLEC. When referencing memory mapped to the USER system, ICE-85 expects an acknowledge signal from the user memory. When referencing memory mapped into the IN-TELLEC, ICE-85 expects two such acknowledge signals, one from the user system and one from the Intellec.

If all memory accessed by a user program is of type USER, physical memory must be present in the selected address locations. In this case the actual memory device will generate the needed acknowledge signal for read and write operations, (except for on-board PROM; see below).

If all memory accessed by a user program is of type INTELLEC, no physical memory need be present in the user system. The necessary acknowledge signals from the user system will be properly generated if pin 116 is connected to ground.

If some of the memory is USER and some is INTELLEC, a number of problems arise. First, the grounding of pin 116 to handle INTELLEC mapped memory (see preceding paragraph) will prevent proper operation of any physical memory in the

user system, since the acknowledges will be immediately generated upon all read or write requests. These extra acknowledges will cause the system to act as if the data were already present, when in fact it is not.

The only solution is to use the iSBC 80/30 10-microsecond timeout feature. This feature delays acknowledges of access to INTELLEC memory by 10ms. Clearly, this significantly decreases system throughput, since memory access normally requires less than 1 μs. To activate the timeout feature, connect pins 115 - 116 (disconnect pin 116 from ground before making this connection).

A final problem concerns the use of on-board PROM. If you wish to access the memory locations normally found in PROM, without actually using PROMs, how do you properly load the code? The solution depends on the type of memory mapping used. If memory is mapped to the USER system, the PROMs can be disabled by connecting pins 27 - 4, forcing the iSBC 80/30 to go off-board to find this portion of memory. Then, by placing bus memory at location zero, you force the 80/30 to use off-board RAM instead of PROM. When memory is mapped to the Intellec, either the timeout feature of the immediate acknowledge solutions described above may be used.

iSBC 80/10

Terminal Interface

Refer to "Terminal Interface" in the iSBC 80/20 section of this appendix.

iSBC 80/10 Interrupts

The RMX/80 Terminal Handler uses interrupts 6 and 7 for the READ and WRITE interrupts, respectively. The RXRDY and TXRDY signals from the USART (8251 Programmable Communications Interface) are used to generate these interrupts. The time base must be supplied via external hardware such as the iSBC 104 combination expansion board; refer to Appendix G for further information and guidelines.

iSBC 80/10 Baud Rate Selection

The effective baud rate for terminal input is determined by the wire-wrap jumper connection selected in accordance with baud rate selection information in the iSBC 80/10 and iSBC 80/10A Hardware Reference Manual, 9800230.

If the user program specifies a baud rate of 9600 (by setting RQRATE to 7), RMX/80 sets the baud rate factor (see Hardware Reference Manual) to 16. For any other value of RQRATE, RMX/80 sets the baud rate factor to 64, and the jumper connection selected in accordance with the Hardware Reference Manual should correspond to the appropriate baud rate for the factor.

ICE-80 Mode Selection Switch

A special switch must be installed on the System 80/10 to handle properly the acknowledge signals generated for I/O, bus and memory acknowledges. The switch is a 3-position single-pole, double-throw, center-off switch whose functions are labeled LOAD, OFF and RUN.

The LOAD position allows ICE-80 to load code into the iSBC 80/10 which has memory mapped physically into the Intellec development system. The ICE-80 module outputs the memory address and data to both the iSBC and MDS systems

and must get an acknowledge from both systems. Otherwise, ICE-80 reports an error 67 (250 msec time out), which aborts the load process. When loading code into mapped memory, it is common to have no equivalent memory in the iSBC or for that memory to be ROM. In either case, this gives no acknowledge. A pseudo-acknowledge can be created by grounding pin 52, which gives an immediate acknowledge of the iSBC memory access.

The OFF position of the switch is useful for normal operation of the iSBC 80/10 without ICE-80 and with its own full complement of ROM and RAM.

The RUN position of the switch is arrived at by removing the ground from pin 52 (turning the switch to the center-off or position opposite LOAD).

If the system is in an intermediate stage of development in which part of the memory used is on-board or off-board RAM and part is mapped into the Intellec development system via ICE-80 (i.e., neither all mapped nor all physical memory), additional hardware modifications are necessary. These are described in the following section.

Note that the Disk File System services will not operate on a system in which all memory is mapped via ICE-80. Sufficient off-board RAM must be available on the bus for the direct memory access (DMA) operations performed by DFS.

Additional Hardware Modifications for ICE-80

If the memory in your system is a combination of physical memory (on- and/or off-board RAM) and mapped memory, the immediate acknowledge generated by grounding pin 52 (via LOAD position of mode selection switch) cannot be used. In this case the switch should be set to RUN and additional hardware added to the system to generate the required acknowledge signals. Physical memory must be present to occupy all of the address space mapped into the Intellec system via ICE-80.

As an example, assume that the physical memory in your system consists of 4K of PROM (addresses 0-4K) and 1K of RAM (addresses 15-16K) on an iSBC 80/10 board, and 16K of controller-addressable RAM on an iSBC 016 memory expansion board (addresses 48-64K); and assume that you are mapping the address space from 0-8K and 15-16K into an Intellec system via ICE-80. No physical memory occupies the address space from 4-8K. Here are two ways you could provide hardware to supply the required acknowledge signals:

a.   Include another iSBC 016 board in your system, jumpering it to occupy addresses 0-16K. However, note that this memory must not be used for data. This is because overlapping acknowledge signals from the 80/10 board and the auxiliary memory board, occurring when memory locations in the range 0-4K or 15-16K are addressed, may cause the data to be unreliable.

b.   Jumper memory from an iSBC 104 or 108 expansion board to occupy addresses 4-8K. This memory may be used for data, since there is no overlap in hardware addresses.

Accessing Memory in an iSBC 80/10 Using ICE-80

Refer to "Accessing Memory in an iSBC 80 System Using ICE-80" in the iSBC 80/20 section of this appendix.

iSBC 80/10 Wire-Wrap Jumpers

The iSBC 80/10 is shipped from the factory with the following wire-wrap jumpers in place (see the SBC 80/10 and SBC 80/10A Single Board Computer Hardware Reference Manual, Chapter 4, Order Number 9800230F or the System 80/10 Microcomputer Hardware Reference Manual, Chapter 3, Order Number 9800316B).

50 - 51	54 - 55	27 - 29
40 - 41	15 - 16	23 - 24
46 - 47	19 - 20	1 - 2
44 - 45	33 - 34	62 - 64
4 - 8	35 - 36	61 - 63
56 - 57	38 - 39	25 - 26
48 - 49	30 - 31	13 - 14

Table F-1 details the changes that must be made to allow the Demonstration System to operate on the iSBC 80/10. (This will allow interrupt-driven I/O on the 8251 USART at 2400 baud.)

Table F-1. iSBC 80/10 Jumper Changes

Baud Rate	Original Jumper Connection					
	1-2	4-8	15-16	19-20	38-39	56-57
110	2-3	no change	16-17	19-21	37-38	no change
2400	2-3	4-11	16-17	19-21	37-38	remove
any other	2-3	*	16-17	19-21	37-38	remove

*   Refer to baud rate selection information in the iSBC 80/10 Hardware Reference Manual.

If an iSBC 104 combination expansion board is being used to generate a time base (see Appendix G), the I/O address for the interrupt mask register on the iSBC 104 must be set to base address D0. This is accomplished by adding a wire-wrap jumper between pins 1 and 4 in block S2.

In an iSBC 80/10 system an external clock interrupt (e.g., the 1-ms clock on the iSBC 104, 108, 116, or 517) must be supplied to provide timing functions to the system. RMX/80 (see Appendix I) provides default time base service routines specific to the use of any of the abovementioned boards as an external clock source; however, another clock source may be used instead.

A supplemental interrupt polling routine (RQTPOL) is provided in the Demonstration System (see Appendix I) for the clock signal, and level 1 is reserved for its use. Given that the iSBC 104 is set up to generate interrupts at the rate of 1 per millisecond, the RMX/80 clock logic for the iSBC 80/10 counts 50 ticks before updating the Delay List, thus maintaining the 50-ms time quantum as supplied in the iSBC 80/20 and 80/30. Note, however, that this default count value of 50 can be overridden. This is because the iSBC 80/10 version of RMX/80 uses a user-defined public address variable, RQTCNT, which defines the number of clock 'ticks' in a system time unit. For example, if a clock that generates interrupts every 25 milliseconds is being used, the following statement will provide the clock tick value:

    DECLARE RQTCNT ADDRESS PUBLIC DATA (2);

If RQTCNT is not supplied, it will be defaulted to 50 from the UNRSLV.LIB file. Note that other interrupts may share level 1 only if code is added to RQTPOL to handle them.

Choosing a Clock Source

Several things should be considered in choosing a clock source for your iSBC 80/10 system.

First of all, it is required that an interrupt from the clock be able to be sensed by the software. This is necessary so that the interrupt polling routines can determine whether the clock generated an interrupt.

Second, it should be noted that the frequency of the external clock has a direct impact on the amount of processor time available to the application code. It is strongly recommended, therefore, that a clock interrupt period of less than one millisecond be avoided, and a lower-frequency time base is desirable. Overhead is minimized when the number of clock interrupts per system time unit is minimized. As an example, the fact that the iSBC 104 has no programmable counters makes it necessary for the processor to service directly every clock pulse as an interrupt. At the rate of one clock pulse every millisecond, approximately 10% to 15% of the processor time will be used up in interrupt processing.

High clock frequency may actually cause clock interrupts to be missed. This can occur when processing of one clock interrupt requires the transfer of a task from the Delay List to the Ready List, since during part of this transfer all interrupts are disabled. If a second clock interrupt occurs at such a time and the clock interrupt is not latched (see next paragraph), the second interrupt will be missed. If the clock interrupt is latched and a second and third clock interrupt occur during such a transfer, the third clock interrupt will be missed. A clock interrupt will also be missed if it occurs during the execution of a task with software priority higher than or equal to 128, since the iSBC 80/10 version of RMX/80 disables all interrupts while such a task is running.

Third, it is recommended that the clock interrupt be latched and that it be resettable by the software. Latching capability will minimize the chance of missing an interrupt (see paragraph above). A software-resettable latch will allow you to avoid the possibility of the same timer interrupt being serviced (and therefore counted) twice.

Fourth, since RMX/80 requires clock interrupts to be present only when one or more tasks have requested delays, efficiency can be increased if the timer interrupt can be masked off or otherwise disabled when not in use.

Timing Variables for Disk File System

The Disk File System references two EXTERNAL variables pertaining to timing features. For the iSBC 80/20 and 80/30, whose clock periods are fixed, the DFSUNR.LIB default values for these variables should be used, and the user need not be concerned with them. However, for the iSBC 80/10 (unless a 50-ms System Time Unit is used), these variables must be declared PUBLIC and assigned values corresponding to the period of the clock being used.

RQTOV specifies the number of System Time Units DFS is to wait before returning an error code 43 (drive timeout) if a drive being accessed does not respond. The DFSUNR.LIB default is 200 System Time Units, or 10 seconds. For iSBC 80/10 systems, it is recommended that a value be assigned to RQTOV such that the timeout period is approximately 10 seconds. For instance, if you are using a System Time Unit of 20 ms, the recommended value for RQTOV is 500.

RQMOTM is used to determine the time delay supplied by the MOTOR ON operation of the DISKIO service, which is required when using mini-size diskette drives. Like RQTOV, this value specifies a number of System Time Units. The DFSUNR.LIB default is 20 System Time Units, or 1 second. For an 80/10 system, you should assign the value that is equivalent to 1 second in your system. For example, if your System Time Unit is 20 ms, you should initialize RQMOTM to 50.

Time Base Service Operations

For an iSBC 80/10 system, you must provide routines for certain time base service operations referenced by RMX/80. Specifications for these operations are given later in this section. Note that although the specifications are given in the same format as the specifications of RMX/80 Nucleus operations in Chapter 2, these operations are not part of the Nucleus and must be supplied by the user. The UNR-SLV.LIB default for RQCLKI and RQINTI is a null procedure that simply returns. No default routines are provided for RQSTPC, RQSTRC, or the timer polling routine (called RQTPOL in the Demonstration System).

The iSBC 80/10 version of the RMX/80 Demonstration System provides versions of RQINTI, RQSTPC, RQSTRC, and RQTPOL for use with the iSBC 104, 108, 116, or 517 as an external clock source. (RQCLKI is not required when one of these boards is used.)

Specifications

RQCLKI (Clock Initialize) Operation.

Function. This procedure should perform any operations necessary to initialize the hardware that is being used to generate the clock interrupt. RQCLK1 is called from within a critical region (a "protected" section of code which executes with all interrupts disabled) in the RMX/80 START module.

Format.

    RQCLKI: PROCEDURE PUBLIC;
              .
              .
              .
              END RQCLKI;

Description. RQCLKI takes no parameters and need not be reentrant.

This procedure must not enable interrupts or send messages to any exchanges that it does not itself create, since the exchanges specified in the Initial Exchange Table have not been created when this procedure is called. The clock hardware should be initialized so that it does not generate interrupts until RQSTRC is called. If this is not possible, a software flag mechanism should be used. (See the description of the RQSTRC operation for further information.)

If the user is providing the clock interrupt with a programmable counter, this routine would do the necessary programming to initialize the counter.

RQINTI (Interrupt Initialize) Operation.

Function. This procedure should perform any operations necessary to initialize the interrupt hardware environment of the iSBC 80/10. RQINTI is called from within a critical region in the RMX/80 START module.

Format.

    RQINTI: PROCEDURE PUBLIC;
              .
              .
              .
              END RQINTI;

Description. RQINTI takes no parameters and need not be reentrant.

This procedure must not enable interrupts or send messages to any exchanges that it does not itself create, since the exchanges specified in the Initial Exchange Table are not created when this procedure is called. It is intended that this procedure be used only for initialization of external hardware and/or setting of polling routine addresses via the RQSETP facility.

The Demonstration System version of this procedure, designed for use with the clock on the iSBC 104, 108, 116, or 517 expansion board, resets the onboard USART (8251 Programmable Communications Interface chip), calls RQSETP to set up RQTPOL (see below) as the pseudo-level 1 interrupt procedure, and sets the expansion board mask to mask off all its interrupts. (In the process it resets the expansion board clock flip-flop.)

RQSTPC (Stop Clock) Operation.

Function. This procedure should perform the operations necessary to stop the clock interrupt. It is called by the RMX/80 Nucleus when the time base is no longer required — i.e., when no more tasks are requesting timed waits.

Format.

    RQSTPC: PROCEDURE PUBLIC;

             .

             .

             .

    END RQSTPC;

Description. RQSTPC takes no parameters and need not be reentrant.

This procedure should mask off the clock interrupt, if possible. If this cannot be done, a software flag mechanism can be used. See the description of RQSTRC for more information.

The Demonstration System version of this procedure performs the operations necessary to mask off the clock interrupt when the iSBC 104, 108, 116, or 517 is being used to generate this interrupt (and nothing else).

RQSTRC (Start Clock) Operation.

Function. This procedure should perform any operations necessary to enable the clock interrupt. It is called by the RMX/80 Nucleus when the time base is required — i.e., when a task makes a timed wait request.

Format.

    RQSTRC: PROCEDURE PUBLIC;

             .

             .

             .

    END RQSTRC;

Description. RQSTRC takes no parameters and need not be reentrant.

If the clock interrupt cannot be masked off or otherwise stopped, a software flag mechanism can be used. In this case the RQCLKI routine should initialize a flag to a "stopped" state, RQSTRC should set the flag to the "started" state, and RQSTPC (see above) should set it to the "stopped" state. This flag should be tested by the level-one polling routine. It should call the procedure RQCTCK if and only if this flag is in the "started" state. It is very important that RQCTCK not be called when the flag is in the "stopped" state. The flag mechanism described above will not be necessary if the clock interrupt is disabled by RQSTPC is such a way that no clock interrupts are generated, and if the level-one interrupt routine accurately can determine whether a particular interrupt was caused by the clock hardware.

The Demonstration System version of this procedure performs the operations necessary to enable the clock interrupt when the iSBC 104, 108, 116, or 517 is being used to generate this interrupt (and nothing else).

RQTPOL (Timer Poll) Operation.

Function. This procedure serves as one of the user-supplied interrupt polling routines (see "iSBC 80/10 Interrupts" in Chapter 2). It is used by the Demonstration System to recognize the Clock interrupt from the iSBC 104, 108, 116, or 517 board.

Format.

```
RQTPOL:   PROCEDURE BYTE PUBLIC;
             .
             .
             .
          RETURN 0FFH;        /* TRUE */
             .
             .
             .
          RETURN 00H;         /* FALSE */
             .
             .
             .
          END RQTPOL;
```

Description. RQTPOL takes no parameters and need not be reentrant. The procedure must return a boolean value (TRUE = 0FFH or FALSE = 00H).

This procedure is provided by the Demonstration System as the level-one polling routine and serves as the timer poll. The Demonstration System version of RQINTI calls RQSETP to establish this correspondence. Whenever RQTPOL determines that the iSBC 104, 108, 116, or 517 clock hardware has generated an interrupt, it calls RQCTCK.

If you wish to have more than one interrupt source share pseudo-level 1, you should provide your own polling routine via RQSETP (see Chapter 2). This routine should call RQCTCK whenever a clock interrupt is generated; it should also reset the clock interrupt. Your timer polling routine may have a name of your own choosing; since the name is not referenced by RMX/80, it need not be RQTPOL.

Sample Timing Routines

The following two examples are written for use with the clock on the iSBC 104, 108, 116, or 517 expansion board. Example 1 consists of the timer routines in the Demonstration System, which make use of the hardware interrupt enable/disable feature of the expansion boards. Example 2 uses software to logically enable and disable the clock interrupt. This type of logical enable/disable must be performed if your external clock source does not have a hardware enable mask.

Example 1

```
/* MODULE: SBC104
LAST CHANGED: 21 NOVEMBER 1977
THIS MODULE CONTAINS THE ISBC 104 SPECIFIC PROCEDURES FOR HANDLING
THE TIMER INTERRUPTS, SETTING THE INTERRUPT MASK, ETC.
*/
SBC104:DO;
$NOLIST
$INCLUDE (:F2:COMMON.ELT)
$INCLUDE (:F2:EXCH.ELT)
$INCLUDE (:F2:8010.ELT)
$INCLUDE (:F2:IED.ELT)
$INCLUDE (:F2:SYNCH.EXT)
$INCLUDE (:F2:INTRPT.EXT)
$LIST
```

```
RQCTCK:  PROCEDURE EXTERNAL;
END RQCTCK;

/* PUBLIC PROCEDURE:  RQTPOL
   THIS PROCEDURE IS THE TIMER POLLING ROUTINE FOR THE ISBC 104 BOARD.
*/

RQTPOL:  PROCEDURE BYTE PUBLIC;

IF (INPUT(0D0H) AND 01H) THEN
DO;
    /* THIS IS YOUR TIMER INTERRUPT
    */
    OUTPUT(0D2H) = 0;
    CALL RQCTCK;
    RETURN TRUE;
END;
RETURN FALSE;

END RQTPOL;

/* PUBLIC PROCEDURE:  RQINTI
THIS PROCEDURE IS THE INTERRUPT INITIALIZATION ROUTINE FOR THE ISBC 104.
*/

RQINTI:  PROCEDURE PUBLIC;

/* RESET THE USART TO PREVENT STRAY LEVELS FROM CAUSING UNWANTED
   INTERRUPTS
*/
OUTPUT(USART$CONTROL);
OUTPUT(USART$CONTROL);
OUTPUT(USART$CONTROL);
OUTPUT(USART$CONTROL) = 0;
OUTPUT(USART$CONTROL) = USART$RESET;
OUTPUT(USART$CONTROL) = STOP$1 OR CL8 OR RATE$64X;
OUTPUT(USART$CONTROL) = RTS OR ERROR$RESET OR DTR;
OUTPUT(USART$OUT) = 0FFH;
IF INPUT(USART$IN) THEN;

CALL RQSETP(.RQTPOL,1);
OUTPUT(0D3H) = 0H;  /* SET INTERRUPT MASK */

END RQINTI;

/* PUBLIC PROCEDURE:  RQSTRC
THIS PROCEDURE IS THE USER DEFINED PROCEDURE FOR TURNING ON THE
REAL TIME CLOCK ON THE ISBC 104 BOARD. THE CLOCK IS TURNED ON BY ENABLING
THE INTERRUPT MASK BIT ASSOCIATED WITH THE TIMER INTERRUPT. THE
TIMER IS ALSO RESET WHEN THE NEW INTERRUPT MASK IS LOADED.
*/

RQSTRC:  PROCEDURE PUBLIC;

OUTPUT(0D3H) = 01H;

END RQSTRC;

/* PUBLIC PROCEDURE:  RQSTPC
THIS PROCEDURE IS THE USER-CODED ROUTINE THAT DISABLES THE TIMER INTERRUPT.
```

```
*/

RQSTPC:  PROCEDURE PUBLIC;

OUTPUT(0D3H) = 0;

END RQSTPC;

END;
```

Example 2

```
/* MODULE: SBC104
LAST CHANGED: 21 NOVEMBER 1977
THIS MODULE CONTAINS THE ISBC 104 SPECIFIC PROCEDURES FOR HANDLING
THE TIMER INTERRUPTS, SETTING THE INTERRUPT MASK, ETC.

*/
SBC104:DO;
$NOLIST
$INCLUDE (:F2:COMMON.ELT)
$INCLUDE (:F2:EXCH.ELT)
$INCLUDE (:F2:8010.ELT)
$INCLUDE (:F2:IED.ELT)
$INCLUDE (:F2:SYNCH.EXT)
$INCLUDE (:F2:INTRPT.EXT)
$LIST

RQCTCK:  PROCEDURE EXTERNAL;
END RQCTCK;

DECLARE TIMER$ENABLED BYTE;

/* PUBLIC PROCEDURE:  RQTPOL
  THIS PROCEDURE IS THE TIMER POLLING ROUTINE FOR THE ISBC 104 BOARD.
*/

RQTPOL:  PROCEDURE BYTE PUBLIC;

IF (INPUT(0D0H) AND 01H) THEN
DO;
    /* THIS IS YOUR TIMER INTERRUPT

    OUTPUT(0D2H) = 0;
    IF TIMER$ENABLED THEN
    CALL RQCTCK;
    RETURN TRUE;
END;
RETURN FALSE;

END RQTPOL;

/* PUBLIC PROCEDURE:  RQINTI
   THIS PROCEDURE IS THE INTERRUPT INITIALIZATION ROUTINE FOR THE ISBC 104.

*/

RQINTI:  PROCEDURE PUBLIC;
```

```
/* RESET THE USART TO PREVENT STRAY LEVELS FROM CAUSING UNWANTED
   INTERRUPTS
*/
OUTPUT(USART$CONTROL);
OUTPUT(USART$CONTROL);
OUTPUT(USART$CONTROL);
OUTPUT(USART$CONTROL) = 0;
OUTPUT(USART$CONTROL) = USART$RESET;
OUTPUT(USART$CONTROL) = STOP$1 OR CL8 OR RATE$64X;
OUTPUT(USART$CONTROL) = RTS OR ERROR$RESET OR DTR;
OUTPUT(USART$OUT) = 0FFH;
IF INPUT(USART$IN) THEN;

CALL RQSETP(.RQTPOL,1);
OUTPUT(0D3H) = 01H;  /* ENABLE THE CLOCK INTERRUPTS */
TIMER$ENABLED = FALSE;

END RQINTI;

/* PUBLIC PROCEDURE:  RQSTRC
THIS PROCEDURE IS THE USER-DEFINED PROCEDURE FOR TURNING ON THE REAL-
TIME CLOCK ON THE ISBC 104 BOARD. THE CLOCK IS TURNED ON BY LOGICALLY
ENABLING THE INTERRUPTS TO CAUSE SOME ACTION OTHER THAN THE
RESETTING OF THE TIMER FLIP-FLOP.
*/

RQSTRC:  PROCEDURE PUBLIC;

TIMER$ENABLED = TRUE;

END RQSTRC;

/* PUBLIC PROCEDURE:  RQSTPC
THIS PROCEDURE IS THE USER-CODED ROUTINE THAT DISABLES THE TIMER
INTERRUPT LOGICALLY, WHICH WILL PREVENT ANY ACTION OTHER THAN THE
RESETTING OF THE TIMER FLIP-FLOP.
*/

RQSTPC:  PROCEDURE PUBLIC;

TIMER$ENABLED = FALSE;

END RQSTPC;

END;
```

A problem that may arise in multitasking systems, including those based on RMX/80, is deadlock — or, as it is sometimes called, the "deadly embrace." This situation occurs when each of a group of tasks is waiting for a resource that is being held by another task in the group. The result is a "circular wait" — the tasks in the group are all waiting for each other, and therefore all will wait indefinitely. The group of tasks is thus effectively "dead."

For example, assume that two tasks, X and Y, in an RMX/80 system each require both the services of a high-speed math board (driven by a user-written handler task) to perform a set of arithmetic operations and analog I/O services provided by a single analog input/output board. The board that performs the arithmetic calculations and the board that performs the analog input and output are both single-user resources — i.e., resources to which only one task can have access at any given time. This exclusive access is ensured by setting up an exchange at which a token message is available when the resource is not in use. A task wishing to use the resource waits at this exchange and receives the token message; when that task is finished with the resource, it sends the token message back to the exchange. Any other task requesting use of the resource must wait at the exchange until the first task returns the token message. (This is the technique discussed in Chapter 3 under "Exclusive Access to Code.")

In our example, let exchange A control access to the math board, and exchange B control access to the analog input/output board. A programmer not concerned with the possibility of deadlock might code the two tasks in the following manner:

Task X (high priority)	Task Y (low priority)
ADRA = RQWAIT (.A,0);	ADRB = RQWAIT (.B,0);
(perform calculations)	(perform analog input)
ADRB = RQWAIT (.B,0);	ADRA = RQWAIT (.A,0);
(perform analog input and output)	(perform calculations)
CALL RQSEND (.B,ADRB);	CALL RQSEND (.A, ADRA);
(perform calculations)	(perform analog output)
CALL RQSEND (.A,ADRA);	CALL RQSEND (.B,ADRB);

If, in the course of processing, task X is waiting at exchange B and task Y subsequently waits at exchange A, each task will be waiting for a resource (a token message) held by the other task, and there will be a deadlock. Note that these tasks

may run correctly for a long period of time before this situation arises; but once it does occur, the two tasks are "dead" forever.

One way to prevent this problem is to write the tasks so that no task ever holds one resource while waiting for another; i.e., each task has access to only one resource at a time. Tasks X and Y could be rewritten as follows:

Task X	Task Y
ADRA = RQWAIT (.A,0);	ADRB = RQWAIT (.B,0);
.	.
.	.
(perform calculations)	(perform analog input)
.	.
.	.
CALL RQSEND (.A,ADRA);	CALL RQSEND (.B,ADRB);
.	.
.	.
ADRB = RQWAIT (.B,0);	ADRA = RQWAIT (.A,0);
.	.
.	.
(perform analog input and output)	(perform calculations)
.	.
.	.
CALL RQSEND (.B,ADRB);	CALL RQSEND (.A,ADRA);
.	.
.	.
ADRA = RQWAIT (.A,0);	ADRB = RQWAIT (.B,0);
.	.
.	.
(perform calculations)	(perform analog output)
.	.
.	.
CALL RQSEND (.A,ADRA);	CALL RQSEND (.B,ADRB);
.	.
.	.

However, in some cases this solution — allowing each task access to only one resource at a time — may not be desirable or even feasible. Another, more generally applicable, solution is to write the tasks so that they request the resources in the same order — for instance, always A first, then B:

Task X	Task Y
ADRA = RQWAIT (.A,0);	ADRA = RQWAIT (.A,0);
.	.
.	.
(perform calculations)	ADRB = RQWAIT (.B,0);
.	.
.	.
ADRB = RQWAIT (.B,0);	(perform analog input)
.	.
.	.
(perform analog input and output)	(perform calculations)
.	.
.	.
CALL RQSEND (.B,ADRB);	CALL RQSEND (.A,ADRA);
.	.
.	.
(perform calculations)	(perform analog output)
.	.
.	.
CALL RQSEND (.A,ADRA);	CALL RQSEND (.B,ADRB);
.	.
.	.

When the tasks are written in this manner, a deadlock situation cannot occur, no matter where tasks A and B are in their processing at any given time.

It can be shown that this method[1] can be extended to any number of tasks, the rule being that all tasks in a group that shares the same set of resources must request those resources in the same order. If some tasks release some resources before requesting others, it suffices to follow this scheme:

1.   First, list all resources in some order. (You may order them in any way that is convenient or efficient in your system.)

2.   When coding tasks that request any of these resources, always follow the rule that a task may never request one resource when it already has access to (i.e., holds the token message for) any resource that precedes the requested resource on your list. For instance, if you list your resources in the order A, B, C, D, a task may not request resource B while it holds the token for A; it may not request C while it holds the token for A and/or B; and it may not request D while it holds the token for A, B, and/or C.

There is a cost associated with this solution to the deadlock problem, in that a task may be required to hold onto a resource much longer than it actually needs it. For instance, in our illustration with tasks X and Y, suppose that the analog input and output operations take a considerable amount of time, while the arithmetic calculations take a relatively short time. The rewriting of task Y to follow the "order of requests" rule requires that task Y hold resource A (the math board) for the entire time needed to perform its analog input and output using resource B.

Note, however, that in such a situation we can recode both task X and task Y to request resource B before requesting resource A. The order (A before B) is probably appropriate if the use of resource B is for short periods of time and/or other tasks require the use of B and not of A.

In systems where this solution proves inefficient, there are other methods to prevent deadlock, or alternatively, to detect and recover from it. Coffman, Elphick, and Shoshani[1] discuss the ordering method described here, plus a number of other solutions.

---

[1]Coffman, E.J., Jr., M.J. Elphick, and A. Shoshani, "System Deadlocks," *Computing Surveys* , Vol. 3, No. 2, June 1971, pp. 67-78.

The RMX/80 Executive diskettes include all software for a Demonstration System which uses the full input-output Terminal Handler, Free Space Manager, and active Debugger, and adds a simple command line interpreter. The command line interpreter supports a few simple commands, entered from the terminal, that allow you to request certain functions from the Free Space Manager. You can use the Debugger to observe what is happening in your system.

The primary purpose of this system is to help you check out your hardware configuration to ensure that it operates correctly with RMX/80. Once you have the Demonstration System running properly, you can proceed to test your application tasks with confidence that any errors found are the fault of the application software, not of your hardware. The Demonstration System is also useful in giving you "hands-on" familiarity with the Terminal Handler, Free Space Manager, and Debugger. It is suggested that all new RMX/80 users load and experiment with the Demonstration System before attempting to install an application system.

Loading and Starting the Demonstration System

The Demonstration System requires less than 21K bytes of code space (less than 20K for iSBC 80/20 and 80/10 versions) and less than 6K bytes of data space. It can run in one of two ways: in Intellec mapped memory via ICE-80 or ICE-85, or in memory in your system. Note that to run the Demonstration System from iSBC, memory expansion boards are required.

For iSBC 80/10-based systems, the Demonstration System supports the use of the clock hardware on an iSBC 104, 108, 116, or 517 combination expansion board, and a terminal with a transmission rate of 2400 baud. If you have an 80/10 system and your terminal's transmission rate is other than 2400 baud, you must set up the system for the proper baud rate. To accomplish this, add a PUBLIC declaration of the proper RQRATE value (see table 4-1) to DEMO.M80 (the assembly language configuration module for the Demonstration System), then re-generate the system with the ISIS-II command SUBMIT :Fn:LOADX (where your RMX/80 Executive diskette is on drive n).

Running in Intellec Mapped Memory via ICE-80 or ICE-85

The RMX/80 Executive diskettes provide an ISIS-II SUBMIT file (LOADX-.CSD) that loads the Demonstration System automatically via ICE-80 or ICE-85. This SUBMIT file is used when you wish to use Intellec memory in place of iSBC memory. The instructions that follow tell how to start up and run the Demonstration System using ICE.

1. Set up your hardware configuration, which must include a terminal for your iSBC system (in addition to the Intellec terminal used by ISIS-II and ICE). For installation instructions, refer to Appendix F and to the appropriate hardware installation manuals for your equipment.

2. Start up your system hardware and the Intellec system as directed in the appropriate hardware operator's manuals.

3. Insert an ISIS-II system diskette (including ICE-80 or ICE-85) in drive 0 of your Intellec system, and your RMX/80 Executive diskette in drive 1.

4. Start ISIS-II as directed in the ISIS-II User's Guide (9800306).

5. For iSBC 80/20 and 80/10 systems, set the ICE-80 mode selection switch on your system to LOAD.

6. Give the ISIS-II command SUBMIT :F1: LOADX. The system will respond by displaying the commands in the SUBMIT file as they are executed. Wait a few minutes until the line *LOAD :F1:DEMO appears, followed by a line ending in an E prompt.

7. · For iSBC 80/20 and 80/10 systems, set the ICE-80 mode selection switch to RUN.

8. Give the ICE-80 or ICE-85 command GO FROM 0.

9. If your system is an iSBC 80/20 or 80/30, initialize the baud rate on your RMX/80 terminal by typing a series of U's, as described in Chapter 4 under "Automatic Baud Rate Search." When the rate is properly initialized, the following line will be displayed (followed by a carriage return):

   RMX/80 COMMAND LINE INTERPRETER, Vn.n

   where n.n is the version number of your RMX/80 software. On an 80/10-based system, this line should be displayed automatically after step 8 is performed, without any need to initialize the baud rate, provided you have set up the Demonstration System for the proper baud rate as directed at the beginning of this section.

10. You may now enter Demonstration System commands, which are described later in this appendix, to communicate with the Free Space Manager. Note that since the Demonstration System runs under the control of the Terminal Handler, all Terminal Handler control character commands are active, and you may invoke the Debugger as usual. (Refer to Chapters 4 and 6 for details.)

11. When you are finished with the Demonstration System, you may use the Intellec system interrupt 4 to return to ICE-80, the ESCape key to return to ICE-85, interrupt 1 to return to ISIS-II, or interrupt 0 to invoke the Intellec monitor program.

Running in PROM

To run the Demonstration System in PROM, proceed as follows:

1. Program your PROMs with the code from the file DEMO on the RMX/80 Executive diskette.

2. Install the PROMs starting at address 0 on your system.

3. Set up your hardware configuration, which must include a terminal for RMX/80, in accordance with Appendix F and the appropriate hardware installation manuals for your equipment.

4. Start up your system hardware and press RESET.

5. For iSBC 80/20 and 80/30 systems, initialize the baud rate as in step 9 of the procedure given for running in mapped memory.

6. Step 10 of the mapped memory operating procedure now applies.

Demonstration System Commands

General Instructions

The Demonstration System provides four commands for communication with the Free Space Manager. To enter a command, type the command name, a space, the parameter, if any, and a carriage return. As mentioned earlier, the Demonstration System operates under the Terminal Handler, so all Terminal Handler control commands are active. The Debugger may be invoked with a control-C.

The parameters in the GET and PUT commands may be entered in decimal, hexadecimal, binary, or octal; non-decimal numbers must include the appropriate suffix (H for hexadecimal, B for binary, O or Q for octal) following the rightmost digit. The Demonstration System gives responses in hexadecimal (without the H suffix).

When the system is started or RESET, the Demonstration System initializes the Free Space Manager, so memory is immediately available for allocation.

Command Descriptions

GET Command. GET allocates a block of RAM from the Free Space Manager. The format is as follows:

   GET size

The GET command name may be abbreviated as G. *Size* must specify the length (in bytes) of the message to be allocated, which must be a multiple of four bytes. If no parameter is given, a size of 8 (the minimum allocation) is assumed.

If the Free Space Manager is able to make the allocation, the system displays "FS$REQ" followed by the address and length of the allocated message in hexadecimal. If the allocation request is too large and cannot be satisfied, the display indicates "FS$NAK" followed by the length of the largest available block of RAM.

PUT Command. PUT returns a block of RAM to the Free Space Manager's pool. The format is as follows:

   PUT message-address

The PUT command may be appreviated as PU. *Message-address* specifies the address (usually in hexadecimal; be sure to include the 'H' suffix) of the message to be returned to the Free Space Manager. If the address is invalid, or if no address is given, the Demonstration System does not respond.

POOL Command. POOL displays the location and size of each block of free RAM in the Free Space Manager's pool. The format is as follows:

   POOL

The POOL command may be abbreviated as PO. The command takes no parameters.

For each block of free RAM, the system displays the word "POOL", followed (in hexadecimal) by the starting address and length (in bytes) of the block. If the pool is empty, nothing is displayed.

Two things should be noted about the Demonstration System's displays for the POOL command. First, two consecutive iterations of POOL often return responses that add up to the same amount of free RAM, but in different locations. This occurs because the Demonstration System's Command Line Interpreter task uses the Free Space Manager for allocation of a message in which to store the current command being entered from the terminal. Thus the operation of the Demonstration System itself affects the RAM pool. Second, POOL occasionally displays considerably less memory than should be available. This happens when the POOL task preempts the Free Space Manager's merge task (which runs at priority 254) while the merge task is in the process of combining two contiguous blocks of RAM. Since the merge task has temporarily removed the two blocks of RAM from the pool, the POOL command will not display these blocks. If you give the POOL command again, the merge task should be finished, and the correct amount of memory should be displayed.

MESSAGE Command. MESSAGE displays the list of messages allocated by the GET command and not yet returned by the PUT command. The format is as follows:

MESSAGE

The MESSAGE command may be abbreviated as M. The command takes no parameters.

For each currently allocated message, the system displays the text "MSG" followed by the starting address and length of the message. If no RAM is currently allocated to the user, there is no display.

Introduction

This appendix contains a complete, executable RMX/80-based system called RTCLOK (real-time clock). This extended example:

- pulls together a number of RMX/80 concepts and facilities into a unified working system
- is simple enough to be grasped by first-time RMX/80 users
- implements a function that is relevant to many applications
- contains some useful routines that can be borrowed by RMX/80 users

The point of the example is not to show how to implement a real-time clock with RMX/80. Instead, the real-time clock is used as a vehicle to illustrate basic RMX/80 operations: sending and receiving messages, creating exchanges, using the Terminal Handler, and so on.

The rest of the appendix is divided into three sections. The first describes the example system from both external (what it does) and internal (how it does it) points of view. The second section provides some background on how the system was designed and tested; this information may be helpful to you as you develop your own approach to building RMX/80-based systems. The last section contains documentation and source code listings for all the modules in the example.

System Operation

External

RTCLOCK is fundamentally an expensive alarm clock. It uses a CRT terminal to . display time of day and the setting of the alarm once per second like this:

    TIME=09:22:30--ALARM=10:00:00

When the alarm "goes off," three exclamation points are appended to the display line and the terminal's audible alarm is sounded. Commands can be entered from the keyboard to:

- suspend the time display
- change the setting of the clock
- change the setting of the alarm
- request a "menu" display of valid commands

The system is designed to run on an iSBC 80/20 and it determines the CRT's baud rate automatically. It could be modified to run on an iSBC 80/30, or an iSBC 80/10 with additional time base hardware. With a simple change, a hard-copy terminal could be used in place of the CRT.

A typical terminal session illustrating the operation of RTCLOK is shown in figure J-1. Operator entries are shown in boldface, system displays in regular type. A carriage return is indicated by (**CR**).

```
            U    (Terminal Handler baud rate search)

RTCLOK:    VERSION 1.0

ALL TIMES ARE 24-HOUR FORMAT:   HH:MM:SS
      09:22:30 = 9:22:30 AM
      17:30:00 = 5:30 PM
      00:00:00 = MIDNIGHT
      24:00:00 = NO TIME - CLOCK IS OFF


AVAILABLE COMMANDS:
      C = SET REAL-TIME CLOCK
      A = SET ALARM
      ? = DISPLAY THIS COMMAND MENU
ENTERING A "NULL COMMAND" (CARRIAGE RETURN ONLY) SUSPENDS THE
AUTOMATIC TIME DISPLAY.
IF YOU ENTER C OR A, THE SYSTEM WILL PROMPT YOU TO ENTER A TIME.
REMEMBER TO PUT IN COLONS.

TO SHUT OFF THE ALARM, SET IT TO 24:00:00.
      TIME=24:00:00—ALARM=24:00:00       time displays write
      TIME=24:00:00—ALARM=24:00:00       over each other
      TIME=24:00:00—ALARM=24:00:00       on the same line.
(CR)
C(CR)
HH:MM:SS?
09:25:58
      TIME=09:25:59—ALARM=24:00:00
      TIME=09:26:00—ALARM=24:00:00
A(CR)
HH:MM:SS?
09:26:10(CR)
      TIME=09:26:07—ALARM=09:26:10
      TIME=09:26:08—ALARM=09:26:10
      TIME=09:26:09—ALARM=09:26:10
      TIME=09:26:10—ALARM=09:26:10!!!    (bell sounds)
      TIME=09:26:11—ALARM=09:26:10!!!    (bell sounds)
X(CR)
INVALID COMMAND

AVAILABLE COMMANDS:
      C = SET REAL-TIME CLOCK
      A = SET ALARM CLOCK
      ? = DISPLAY THIS COMMAND MENU
ENTERING A "NULL COMMAND" (CARRIAGE RETURN ONLY) SUSPENDS THE
AUTOMATIC TIME DISPLAY.
IF YOU ENTER C OR A, THE SYSTEM WILL PROMPT YOU TO ENTER A TIME.
REMEMBER TO PUT IN COLONS.

TO SHUT OFF THE ALARM, SET IT TO 24:00:00.
```

Figure J-1. RTCLOK Terminal Session

Internal

Figure J-2 provides an overview of RTCLOK's RMX/80 structure. There are three user tasks: ONESEC, CONSOL and UPTIME.
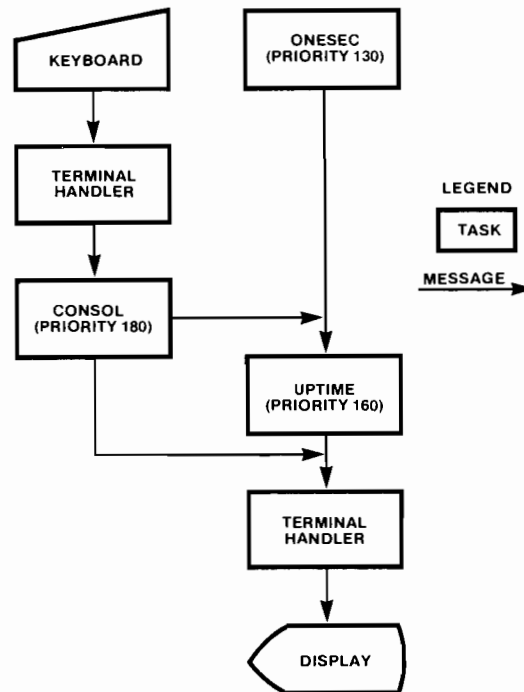
Figure J-2. RTCLOK: Simplified Structure

ONESEC sends a message to UPTIME every second to indicate the passage of time. CONSOL sends messages to UPTIME in response to commands received from the terminal. CONSOL also sends messages to the CRT via the RMX/80 Terminal Handler. UPTIME processes the messages it receives from CONSOL and ONESEC, and as a result, may send messages to the display also.

Figure J-3 illustrates RTCLOK in more detail. Tasks are shown as before; exchanges and message names have been added. Notice that the Terminal Handler is depicted as a "black box" — its own tasks, internal exchanges and messages are not shown. This is partly for clarity, but it also reflects a user task's "view" of the Terminal Handler — what goes on "inside" is of no concern to a user task. Five additional Terminal Handler exchanges are drawn with dashed lines: RQWAKE, RQALRM, RQDBUG, RQL7EX and RQL6EX. These exchanges are not used by RTCLOK, but are shown in the figure because they must be included in the configuration module of any system that uses the Terminal Handler.

ONESEC waits at the WAITX exchange for 20 system time units (50 milliseconds each, one second total). Since no messages are ever sent to the WAITX exchange, ONESEC always receives a system message (TYPE=3H=TIMEOUTTYPE) at the end of one second. It then sends an ADVANCE$TIME message to the UPTIMX exchange, signalling UPTIME to move the clock ahead one second. ONESEC then waits for another second, sends another ADVANCE$TIME message, and so on indefinitely.

CONSOL sends a COMAND message to the Terminal Handler input exchange (RQINPX) to request an input line from the terminal. It waits for a response from the Terminal Handler at the response exchange called CREADX. When the Terminal Handler returns the COMAND message to CREADX, CONSOL calls a routine to process the command contained in the message. CONSOL uses a message called REPLY to provide feedback to the operator. Whenever it sends the REPLY message to RQOUTX (the Terminal Handler output exchange), CONSOL waits at CWRITX until the Terminal Handler has finished writing the message line to the display. After it has processed a command, CONSOL again sends the COMAND message to the RQINPX exchange and waits for the operator to enter the next command. This process continues indefinitely.

CONSOL can send two messages to UPTIME, via the UPTIMX exchange. These are triggered by commands entered by the terminal operator. The ENABLE$DISPLAY message instructs UPTIME to start or stop the automatic time display (message TYPE=140 starts the display and TYPE=130 stops it). The display is suspended upon request from the operator (a carriage return only is entered),
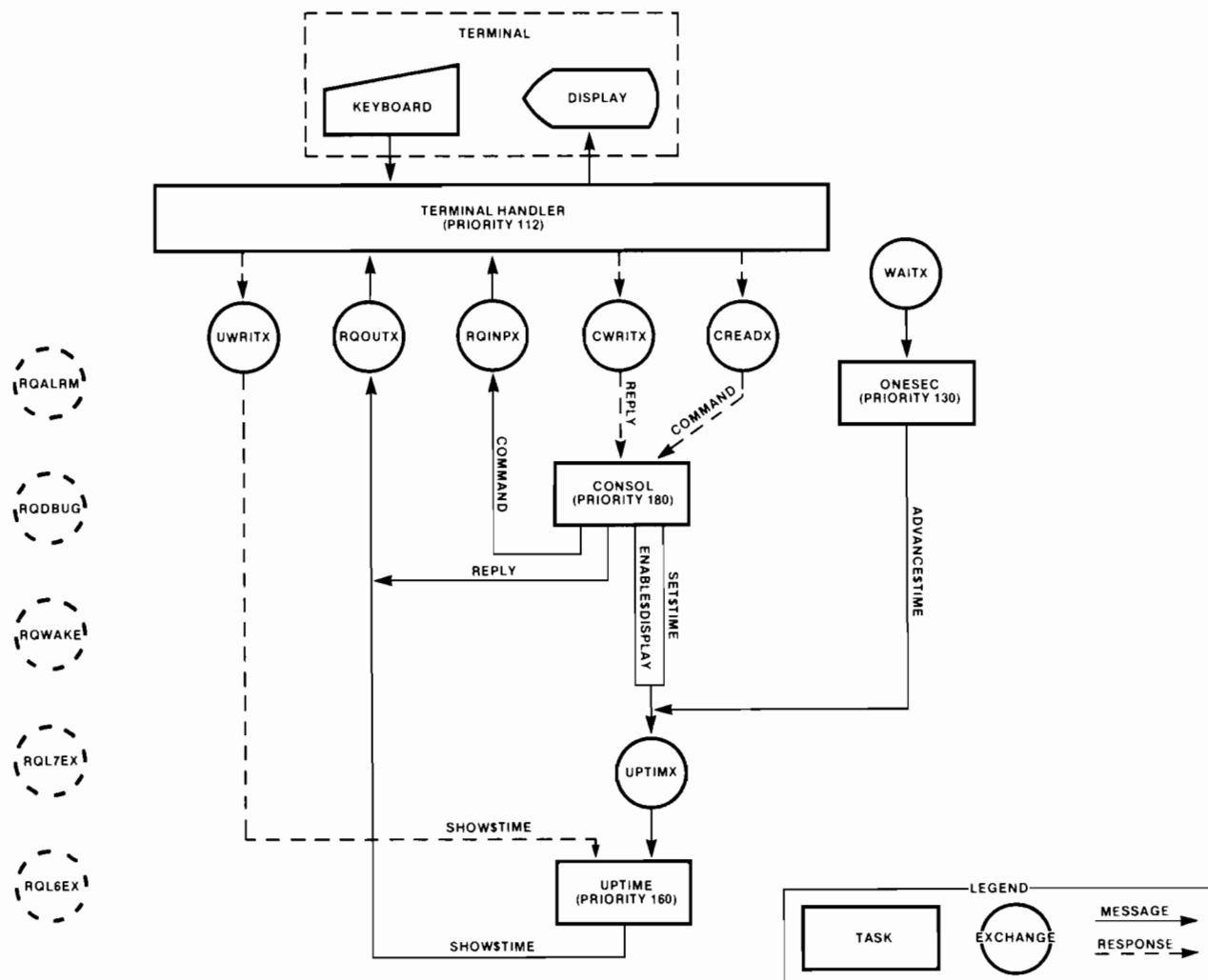
Figure J-3. RTCLOK: System Structure/Message Flow

when the operator is responding to a query from CONSOL, and whenever CONSOL is sending a series of messages to the terminal. Suspending the time display like this prevents different messages from becoming interspersed with each other and with echoed operator entries on the display. The real-time clock keeps running, however, regardless of whether it is being displayed.

The other message that CONSOL sends to UPTIME is SET$TIME. If this message has a TYPE of 110, UPTIME is to change the setting of the alarm to the value contained in the text of the message. If the SET$TIME message's TYPE is 120, UP-TIME is to use the value in the message to set the real-time clock instead.

UPTIME maintains the real-time clock in response to messages received from CON-SOL and ONESEC. UPTIME waits for a message at the UPTIMX exchange. If the message is ENABLE$DISPLAY, UPTIME turns the automatic time display on or off. If the message is SET$TIME, UPTIME sets the alarm or the real-time clock. If the message is ADVANCE$TIME, UPTIME moves the clock ahead one second and, if the display is currently enabled, displays the new time and alarm setting by sending the SHOW$TIME message to the Terminal Handler output exchange RQOUTX. Before sending the message, UPTIME compares the values of the real-time clock and the alarm. If the alarm should be "ringing," UPTIME adds three exclamation points and two BEL characters to the SHOW$TIME message. The BEL characters will cause the terminal's audible alarm to sound, if it has one.

Development Background

RTCLOK was developed in steps, following a conventional sequence:
- defining requirements
- designing the system
- coding the modules
- linking and locating the modules
- testing the system

What "went on" in each of these steps is the subject of this section.

Defining Requirements

RTCLOK's requirements were identified as follows:

Functions
- keep time with one second resolution
- indicate if time >= alarm
- allow clock and alarm to be set from console
- display time and alarm automatically
- suspend automatic time display upon request
- reject invalid operator entries

Events
- operator enters data
- one second passes

Constraints
- simple task/exchange/message structure
- to avoid losing time, a task must be waiting whenever the RMX/80 clock "ticks" (every 50 msec.)

Hardware
- iSBC 80/20
- CRT terminal

Designing the System

The design activity transformed the system requirements into an RMX/80 task/message/exchange structure and a module (program) structure.

First a general design was produced, resulting in the system as depicted in figure J-2. This helped to delineate tasks and their priorities. The system requirements were grouped into logically related clusters that might be implemented as a few coherent RMX/80 tasks. The groups that evolved were:
- interact with the operator via the console (terminal).
- maintain and display the clock.
- wait for one second.

These became the CONSOL, UPTIME (update time) and ONESEC tasks. Why wasn't ONESEC combined with UPTIME? A single task could easily perform the one second wait and still display the clock and alarm settings between clock ticks. A problem might arise, though, if this task also had to process requests to start/stop the display and reset the clock and the alarm. These requests occur unpredictably (asynchronously), usually in response to operator entries. It is true that the operator can only enter a few such requests compared to the system's processing speed, and it is likely that all such requests could be handled between clock ticks. However, this approach would complicate the design and would be limiting. What if other tasks were added to the system that requested time information? It was definitely a question of judgment, but it seemed simpler to separate the "wait for one second" requirement from the "maintain and display time" requirement. In general, allocating different, or loosely related, functions to separate tasks yields the best design.

Priorities were assigned to the three tasks according to the "criticality" of their functions. Clearly, the system would lose its integrity (would not keep accurate time) if it missed a clock tick, so ONESEC was given the highest priority. (In addition, ONESEC takes so little time to execute that its high priority would not interfere with lower priority tasks.) On the other hand, the task that could best afford to be delayed during periods of high loading (as unlikely as this might be in a system as simple as RTCLOK) was CONSOL: operator commands would be relatively infrequent events in the first place, and responses could be delayed for fairly long (say a second or two) periods without unduly inconveniencing the operator. This left UPTIME with a middle priority which seemed appropriate for its needs: having a higher priority than CONSOL insured that UPTIME would be able to "keep up" with the messages sent by CONSOL. Note that difficulty in assigning priorities can be a sign of improper task definition; usually a task needs to be subdivided into tasks with different priorities.

The priorities of all the tasks (ONESEC: 130, UPTIME: 160, CONSOL: 180) are lower than the RMX/80 software priorities that correspond to hardware interrupt levels (0-128). None of the tasks services a hardware interrupt; keeping them out of the hardware interrupt range insures that they will not interfere with tasks like the Terminal Handler that must respond to hardware requests for service.

ONESEC will not be appreciably delayed by the Terminal Handler, even though it has a lower priority. This is because the Terminal Handler works with one character of message text at a time. It relinquishes control of the system (waits) between characters, giving ONESEC the opportunity to run if it is ready.

RTCLOK's exchange structure was largely dictated by the nature of the system's tasks. The RMX/80 Terminal Handler performs all housekeeping involved with terminal I/O. Therefore terminal input is obtained by CONSOL from the RQINPX Terminal Handler exchange. Likewise, output displays generated by CONSOL and UPTIME are sent to the Terminal Handler output exchange RQOUTX. In order for ONESEC to time out at one-second intervals, it needs to wait at an exchange to which no messages are ever sent; the WAITX exchange is dedicated to this single use. Whenever CONSOL or ONESEC want UPTIME to "do something" to the clock, the alarm, or the time display, they send a message to the UPTIMX exchange. Queuing these various types of messages at a single exchange is appropriate for UPTIME because it deals with fairly low message rates. One message per second arrives from ONESEC, and a few messages come from CONSOL each time the operator makes an entry. What if, however, CONSOL were capable of sending bursts of, say, 100 messages per second? ONESEC assumes that UPTIME has processed the previous ADVANCE$TIME message and reuses the RAM area each second. Having to process an intervening burst of messages from CONSOL could well cause UPTIME to miss an ADVANCE$TIME message, thereby causing the clock to lose one second. In a situation like this it would be better to split UPTIME into separate tasks, one to handle messages from CONSOL and one to service messages from ONESEC.

After the general design depicted in figure J-2 was verified against the system requirements and subjected to a mental "simulation" of operation, the design was expanded to the more detailed structure shown in figure J-3. Response exchanges were added to coordinate interaction with the Terminal Handler. A task that sends a message to the Terminal Handler must not modify the message or send it again until the Terminal Handler is through with the message. The Terminal Handler returns the message to the exchange specified in the message's Response Exchange field when it has completed the requested operation. Therefore, CONSOL and UPTIME wait at response exchanges until they receive acknowledgement from the Terminal Handler that the previously sent message can be reused.

Unused, but required, Terminal Handler exchanges were also defined at this time to be sure that they were included in the configuration module.

Finally, messages were named and their contents defined. To keep the number of messages (and consequently RAM requirements) to a minimum, some messages are used for more than one function. For example, the same message (SET$TIME) is used to instruct UPTIME to set the alarm or the clock (the functions are distinguished by different TYPE codes). Using this technique requires special care. With two different "versions" of the same message (i.e., the same RAM area), a second version of a message can overwrite a previous version that has not yet been processed by the receiving task. This cannot happen in RTCLOK because UPTIME runs at a higher priority than CONSOL. Every time CONSOL sends a message to UPTIME, UPTIME becomes the running task and processes the message. Alternatives to this approach include using the Free Space Manager to obtain and release RAM areas for the different versions and using response exchanges to coordinate message reuse.

A module is a segment of code that can be assembled or compiled as an individual unit. Designing RTCLOK's module structure was simple because RMX/80 does not restrict module definition in any way. An RMX/80 task is a public procedure. Multiple tasks can be combined into a single module or a single task may consist of multiple modules. For that matter, the configuration "module" does not even have to be a module. In general, it is good practice to code each task as one or more modules and the configuration module as one module.

RTCLOK was divided into modules as follows:

Task	Modules
ONESEC	ONESEC
UPTIME	UPTIME,NUMOUT
CONSOL	CONSOL,SCANIN

The configuration module was coded as a sixth module called RTCLOK. NUMOUT and SCANIN had already been implemented as modules for another (non-RMX/80) system and were incorporated without change. Note that each of these procedures is called by only one task. If these procedures were to be shared among tasks, software lockouts or reentrant coding as described in chapter 3 would have been required.

Coding the Modules

All modules were written in PL/M-80 with the exception of the configuration module. The assembly language macros supplied with RMX/80 were used to generate this module. Although the choice of assembly language over PL/M-80 in this case was largely a matter of personal preference, using the macros does require fewer lines of code, which usually means fewer chances for error. Errors in the configuration module are generally disastrous to a system and are difficult to diagnose.

The following code was used in RTCLOK's configuration module to prevent unresolved external reference messages for unused interrupt exchanges:

```
;   TIE OFF REFERENCES TO UNUSED INTERRUPT EXCHANGES
;
            DSEG
DUMREF:     DS      2
RQL0EX      EQU     DUMREF
RQL2EX      EQU     DUMREF
RQL3EX      EQU     DUMREF
RQL4EX      EQU     DUMREF
RQL5EX      EQU     DUMREF
            PUBLIC  RQL0EX,RQL2EX,RQL3EX,RQL4EX,RQL5EX
```

So long as an interrupt does not occur at any of the corresponding hardware interrupt levels, RMX/80 does not attempt to reference these exchanges and the unresolved external messages (see chapter 3) can be ignored. However, by not having the message issued routinely every time the system is linked, a "real" unresolved message may be more noticeable if it is issued by the Linker. If an unexpected interrupt for one of these exchanges *does* occur, the exchange *will* be referenced and the result will almost certainly be catastrophic to the system. This is true whether the references are "tied off" as above or not.

Stack space allocations for the tasks were based on the numbers provided by the PL/M-80 compiler, plus the largest stack needed by an RMX/80 Nucleus operation, (see appendix A) plus 24 bytes for RMX/80 to use in a context save.

```
ONESEC(2) + RQWAIT(4) + RMX/80(24) = 30 bytes
UPTIME(10) + NUMOUT(6) + RQWAIT(4) + RMX/80(24) = 44 bytes
CONSOL(8) + SCANIN(4) + RQWAIT(4) + RMX/80(24) = 40 bytes
```

The Debugger Format Task and Scan commands were used during testing to verify that the stacks were not overflowing.

An alternative to setting stack lengths is to make them intentionally too large at first and then use the Debugger to see how much of the space is actually used in execution. A useful aid in this approach is that the Format Task command tells the

number of bytes of the allocated stack space that have not yet been used. Assuming that you can establish worst-case conditions, this command would enable you to determine a stack's maximum possible depth.

ONESEC, CONSOL and UPTIME are patterned after the general RMX/80 "task model." A public procedure is defined with the name of the task. The procedure begins with a section of initialization code. This creates exchanges used by the task (and not already created by earlier users of the exchange), initializes constant portions of messages, sets switches to their initial values, etc. This initialization code is only executed when RMX/80 restarts the system. After executing the initialization code, control falls into an endless loop. This loop contains the main processing logic of the task. CONSOL, for example, waits for a message, processes the message, waits for another message, and so on indefinitely. Execution of this loop will be stopped (usually temporarily) if any of the following happen:

- an interrupt occurs, causing a higher priority task to become ready (this task can in turn cause additional higher priority tasks to become ready)
- a higher priority task in a timed wait times out and thus becomes ready
- the running task sends a message to an exchange where a higher priority task is waiting, thus making it ready
- the running task enters a timed wait or waits at an exchange where no message is queued
- the running task deletes or suspends itself

In the first two cases, the task is preempted by another task, and RMX/80 saves the task's registers on the stack (performs a context save). In the last three cases, the task surrenders control of the processor and the registers are not saved.

Linking and Locating the Modules

After all the modules were compiled/assembled with the DEBUG option (see PL/M-80 Compiler Operator's Manual, 9800300 and ISIS-II 8080/8085 Macro Assembler Operator's Manual, 9800292), the following ISIS-II SUBMIT file was used to link them together with RMX/80 object code and locate the resulting system for testing:

```
LINK    :F1:RMX820.LIB(START), &
        :F1:RTCLOK.OBJ, &
        :F1:SCANIN.OBJ, &
        :F1:ONESEC.OBJ, &
        :F1:CONSOL.OBJ, &
        :F1:UPTIME.OBJ, &
        :F1:NUMOUT.OBJ, &
        :F1:THI820.LIB, &
        :F1:THO820.LIB, &      Terminal Handler
        :F1:ADB820.LIB, &
        :F1:PDB820.LIB,&       Debugger
        :F1:RMX820.LIB, &
        :F1:UNRSLV.LIB, &
        :F1:PLM80.LIB,        TO :F1:RTCLOK.LOD&
                             PRINT(:F1:RTCLOK.LNK) MAP

    LOCATE :F1:RTCLOK.LOD PRINT(:F1:RTCLOK.LOC)&
                         MAP CODE(0) STACKSIZE(0)
```

As will be explained in the next section, not all the tasks were linked in during the first tests. The SUBMIT file shown above was used when the entire system was tested together.

The memory map produced by the Locater is shown below:

```
READ FROM FILE :F1:RTCLOK.LOD
WRITTEN TO FILE :F1:RTCLOK
MODULE START ADDRESS 0000H


START       STOP      LENGTH     REL     NAME

   0H       4A7CH     4A7DH       B      CODE
  28H        2AH        3H        A      ABSOLUTE *CONFLICT*
4A7DH       579DH     DZ13H       B      DATA
579EH       F6BFH     9F22H       B      MEMORY
```

The *CONFLICT* is due to the Debugger's Restart Vector (see chapter 3) and can be ignored. Without the Debugger, RTCLOK's CODE segment is 1DE7H (7655 decimal) bytes long, and the DATA segment is 676 (1654 decimal) bytes.

Testing the System

RTCLOK was tested with the aid of ICE-80 (see In/Circuit Emulator/80 Operator's Manual, 9800185) and the RMX/80 Active Debugger. ICE-80 allowed the program to be executed from Intellec RAM, eliminating the need to program PROMs for the iSBC 80/20. It also allowed controlled execution of the program using breakpoints, and examination of intermediate results in memory. The Debugger was used primarily to monitor stacks; however, it could have been used to set breakpoints and examine memory if ICE-80 had not been available.

The following SUBMIT file was used to load RTCLOK into Intellec memory and begin emulation (execution under the control of ICE-80):

```
ICE80
BASE HEX
XFORM MEMORY 0 INTO 6
XFORM MEMORY 1 INTO 7
XFORM MEMORY 2 INTO 8
XFORM MEMORY 3 INTO 9
XFORM MEMORY 4 INTO 10
XFORM MEMORY 5 INTO 11
XFORM MEMORY 6 TO 15 GUARDED
XFORM IO 0 TO 15 UNGUARDED
LOAD :F1:RTCLOK
```

The first step in testing, however, was to verify that the hardware (iSBC 80/20 and CRT terminal) was set up properly. This was done by loading and executing the Demonstration System supplied with RMX/80 and described in appendix I. Successful execution of this system meant that subsequent difficulties in testing RTCLOK could probably be attributed to software errors, rather than hardware problems.

RTCLOK was tested incrementally, adding one task at a time. This technique simplifies debugging because it usually isolates the source of error to one task, or to the interaction of two tasks. For systems larger than RTCLOK, incremental development can profitably be extended to design and coding as well. This allows different techniques to be tested with quick feedback. A few tasks can be designed, coded and tested, and the knowledge gained from this experience can be used in the development of the rest of the system. Note that the RMX/80 Debugger contains a number of features that complement incremental development.

Incremental testing necessitates making temporary changes to some tasks that reference tasks and data not yet linked into the system. For example, when CONSOL was tested by itself, references to UPTIME were "commented out" like this:

```
/****CALL RQSEND(.UPTIMX,.SET$TIME);****/
```

The configuration module and link statements were also modified as new tasks were added to the test system.

CONSOL was tested first with no other tasks in the system except the Terminal Handler. This verified that the configuration module was correct and that messages were being sent to and received from the terminal properly. (In its first test run, the system did not come up at all. The error was, as it often is in cases like this, in the configuration module.) Messages that CONSOL sent to UPTIME were examined with ICE-80 to insure that they were being constructed properly.

Next UPTIME was linked in so that the interaction between CONSOL and UP-TIME could be checked out. UPTIME's automatic time display was temporarily modified to display the time whenever any message was received rather than just when ADVANCE$TIME arrived from ONESEC (which was not yet in the system).

Finally ONESEC was added. To simplifiy monitoring the system, ONESEC was temporarily changed to issue ADVANCE$TIME every five seconds instead of every second.

Structure Charts and Source Code Listings

The following pages contain a structure chart and the compiled/assembled source code for each of RTCLOK's six modules. A structure chart, as it is used here, is a modification of the concept of flow chart. Because most tasks consist of an initialization phase followed by an endless loop, it is convenient to have a pictorial device that reflects this nicely. In figure J-4, the name of the task appears first, then the initialization phase, after which the "main routine" is first shown, for emphasis, as a single box with "do forever"; the detail of the main routine is seen below the "main routine" box, and it flows from left to right. Thus, the "do forever" part of ONESEC is: wait, send, wait, send, etc. The other tasks, depicted in figures J-5 and J-6, are to be read similarly.
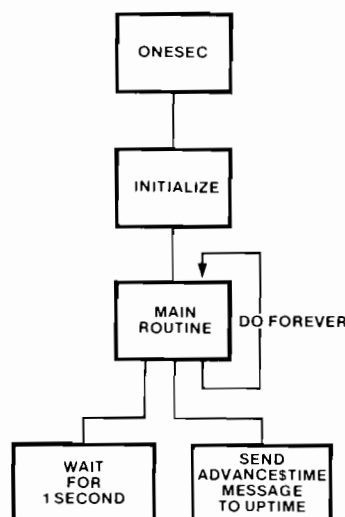
Figure J-4. ONESEC Structure Chart
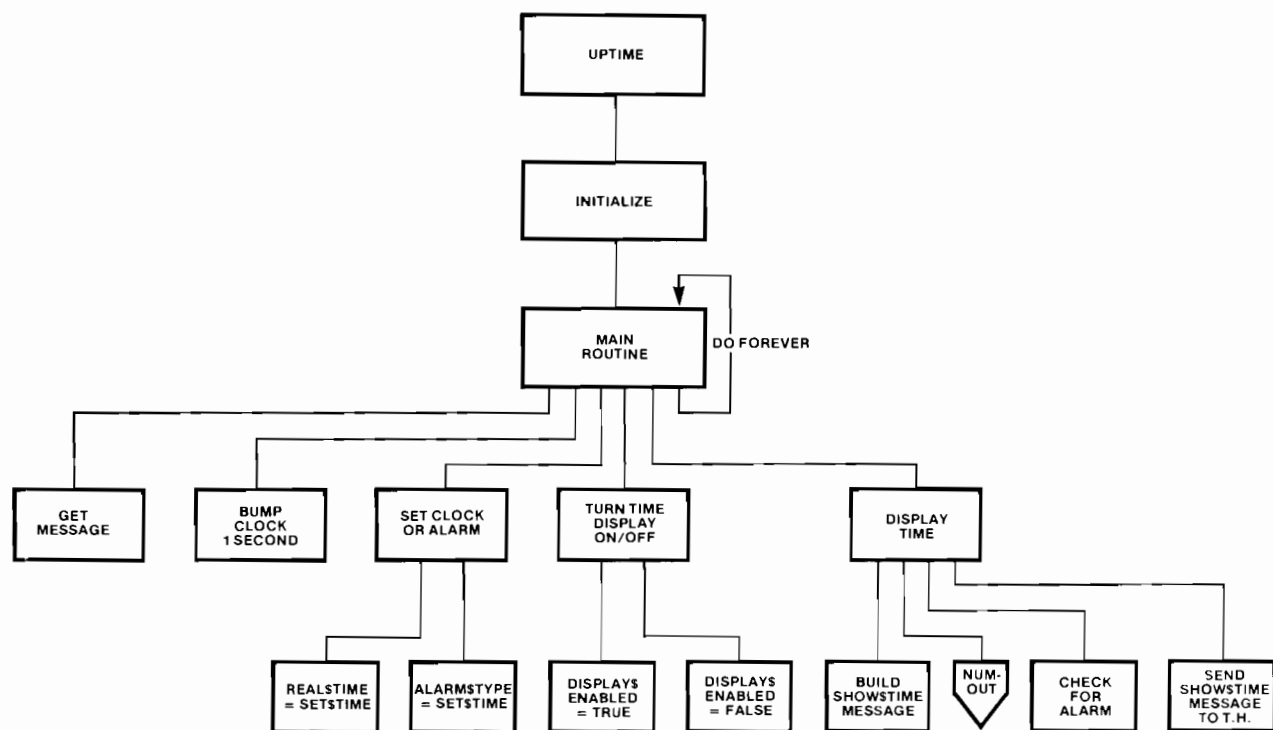
Figure J-5. UPTIME Structure Chart
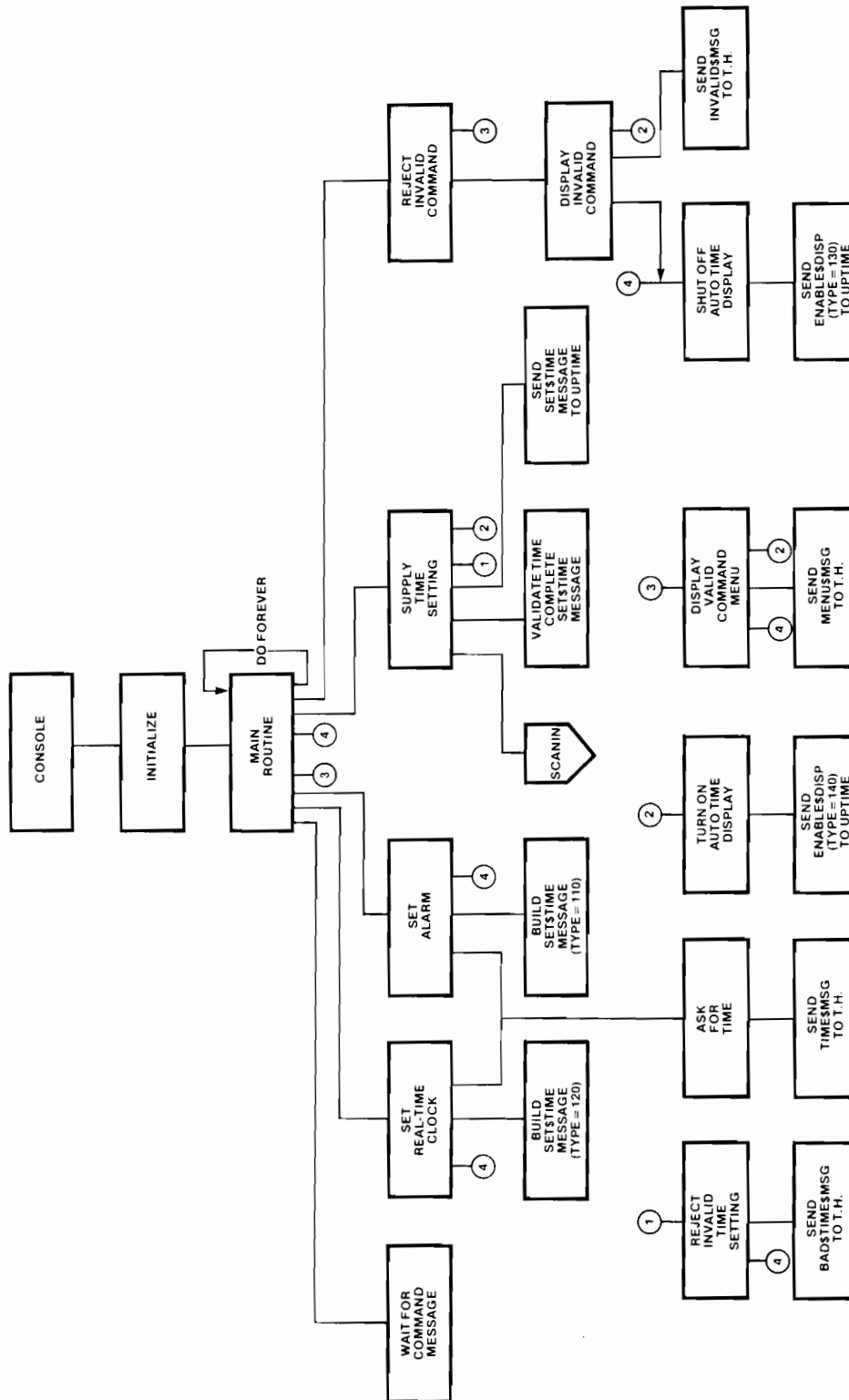
Figure J-6. CONSOL Structure Chart

ASM80.0V3 :F1:RTCLOK.ASM DEBUG

ISIS-II 8080/3085 MACRO ASSEMBLER, V2.0   RTCLOK   PAGE   1
RTCLOK: RMX/80 REAL-TIME CLOCK EXAMPLE

    LOC OBJ   SEQ   SOURCE STATEMENT

                    1 $TITLE('RTCLOK: RMX/80 REAL-TIME CLOCK EXAMPLE')
                    2 ;
                    3 ; CONFIGURATION MODULE FOR REAL-TIME CLOCK EXAMPLE
                    4 ;
                    5 ; HARDWARE ASSUMPTIONS:
                    6 ;   COMPUTER IS ISBC 80/20
                    7 ;   TERMINAL IS CRT
                    8 ;
                    9   NAME RTCLOK
                   10   CSEG
                   11 ;
                   12 ; PICK UP MACRO DEFINITIONS
                   13 ;
              =   14 $INCLUDE (:F1:STD.MAC)
              =   15 ;
              =   16 ; MACRO TO BUILD STATIC TASK DESCRIPTOR (STD) AS
              =   17 ; A COMPONENT OF THE INITIAL TASK TABLE (ITT).
              =   18 ;
              =   19 ; THIS MACRO ACCEPTS 4 PARAMETERS:
              =   20 ;
              =   21 ; (1) THE NAME OF THE TASK ENTRY POINT.
              =   22 ; (2) THE SIZE (IN BYTES) OF THE STACK REQUIRED BY
              =   23 ;   THIS TASK.
              =   24 ; (3) THE PRIORITY OF THIS TASK.
              =   25 ; (4) THE DEFAULT EXCHANGE ADDRESS (IF ANY), TO BE
              =   26 ;   USED WITH THIS TASK.
              =   27 ;
              =   28 STD   MACRO   NAME,STKLEN,PRI,EXCH
              =   29   LOCAL   TSTK
              =   30   EXTRN   NAME
              =   31   IF   NTASK EQ 0
              =   32 ITT.
              =   33   ENDIF
              =   34   DSEG
              =   35 TSTK: DS   STKLEN
              =   36   CSEG
              =   37 $NOGEN NOCOND
              =   38 ADDCHR   MACRO   F1
              =   39 ADDCHR   MACRO   F2
              =   40 ADDCHR   MACRO   F3
              =   41 ADDCHR   MACRO   F4
              =   42 ADDCHR   MACRO   F5
              =   43 ADDCHR   MACRO   F6
              =   44 $GEN
              =   45   DB   'F1&&F2&&F3&&F4&&F5&&F6'
              =   46 $NOGEN
              =   47 ADDCHR   MACRO
              =   48   ENDM
              =   49   ENDM
              =   50   ENDM
              =   51   ENDM
              =   52   ENDM

```
ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0          RTCLOK   PAGE   2
RTCLOK:  RMX/80 REAL-TIME CLOCK EXAMPLE


  LOC  OBJ      SEQ        SOURCE STATEMENT

            =  53        ENDM
            =  54        ENDM
            =  55        IRPC    CHAR,<NAME        >
            =  56        ADDCHR  <CHAR>
            =  57        ENDM
            =  58 $GEN
            =  59        DW      NAME
            =  60        DW      TSTK
            =  61        DW      STKLEN
            =  62        DB      PRI
            =  63        DW      EXCH+0
            =  64        DW      TDBASE+(NTASK*20)
            =  65 NTASK  SET     NTASK+1
            =  66        ENDM
            =  67 #INCLUDE (:F1:XCHADR.MAC)
            =  68 ;
            =  69 ; THIS MACRO WILL BUILD ONE
            =  70 ; COMPONENT OF THE INITIAL
            =  71 ; EXCHANGE TABLE (IET) EACH
            =  72 ; TIME IT IS INVOKED.
            =  73 ;
            =  74 XCHADR MACRO   NAME
            =  75        EXTRN   NAME
            =  76        IF      NEXCH EQ 0
            =  77 IET:
            =  78        ENDIF
            =  79        DW      NAME
            =  80 NEXCH  SET     NEXCH+1
            =  81        ENDM
            =  82 #INCLUDE (:F1:GENTD.MAC)
            =  83 ;
            =  84 ; TASK DESCRIPTOR GENERATION MACRO FOR BUILDING STD
            =  85 ; AND ITT.
            =  86 ;
            =  87 GENTD  MACRO
            =  88        DSEG
            =  89 TDBASE: DS     20*NTASK
            =  90        CSEG
            =  91        ENDM
            =  92 #INCLUDE (:F1:CRTAB.MAC)
            =  93 ;
            =  94 ; THIS MACRO BUILDS THE ROCRTB TABLE.  IT MUST
            =  95 ; BE INVOKED AFTER THE STD AND XCHADR MACROS
            =  96 ; IN ORDER TO WORK PROPERLY.
            =  97 ;
            =  98 CRTAB  MACRO
            =  99        PUBLIC  ROCRTB
            = 100 ROCRTB:
            = 101        DW      ITT
            = 102        DB      NTASK
            = 103        DW      IET
            = 104        DB      NEXCH
            = 105        ENDM
              106 ;
              107 ; CLEAR TEMPS USED BY MACROS
```

```
ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0          RTCLOK   PAGE   3
RTCLOK:  RMX/80 REAL-TIME CLOCK EXAMPLE

  LOC  OBJ        SEQ        SOURCE STATEMENT

                  108 ;
  0000            109 NTASK    SET    0
  0000            110 NEXCH    SET    0
                  111 ;
                  112 ;  TIE OFF REFERENCES TO UNUSED INTERRUPT EXCHANGES
                  113 ;
                  114         DSEG
  0002            115 DUMREF: DS     2
  0000       D    116 RQL0EX  EQU    DUMREF
  0000       D    117 RQL2EX  EQU    DUMREF
  0000       D    118 RQL3EX  EQU    DUMREF
  0000       D    119 RQL4EX  EQU    DUMREF
  0000       D    120 RQL5EX  EQU    DUMREF
                  121         PUBLIC RQL0EX,RQL2EX,RQL3EX,RQL4EX,RQL5EX
                  122 ;
                  123 ;  BUILD INITIAL TASK TABLE ENTRIES
                  124 ,
                  125 ;     (PARAMETERS:  TASK,STACKLENGTH,PRIORITY)
                  126 ;
                  127         CSEG
                  128         STD    CONSOL,42,180
                  129+        EXTRN  CONSOL
                  130+        IF     NTASK EQ 0
                  131+ITT:
                  132+        ENDIF
                  133+        DSEG
  002A            134+??0001: DS     42
                  135+        CSEG
  0000 434F4E53   219+        DB     'CONSOL'
  0004 4F4C         +
  0006 0000     E  230+        DW     CONSOL
  0008 0200     D  231+        DW     ??0001
  000A 2A00        232+        DW     42
  000C B4          233+        DB     180
  000D 0000        234+        DW     +0
  000F DA00     D  235+        DW     TDBASE+(NTASK*20)
  0001            236+NTASK    SET    NTASK+1
                  237         STD    UPTIME,44,160
                  238+        EXTRN  UPTIME
                  242+        DSEG
  002C            243+??0002: DS     44
                  244+        CSEG
  0011 55505449   328+        DB     'UPTIME'
  0015 4D45         +
  0017 0000     E  339+        DW     UPTIME
  0019 2C00     D  340+        DW     ??0002
  001B 2C00        341+        DW     44
  001D A0          342+        DB     160
  001E 0000        343+        DW     +0
  0020 EE00     D  344+        DW     TDBASE+(NTASK*20)
  0002            345+NTASK    SET    NTASK+1
                  346         STD    ONESEC,30,130
                  347+        EXTRN  ONESEC
                  351+        DSEG
  001E            352+??0003: DS     30
```

```
ISIS-II 8080/8085 MACRO ASSEMBLER, V2 0        RTCLOK   PAGE   4
RTCLOK   RMX/80 REAL-TIME CLOCK EXAMPLE

 LOC  OBJ        SEQ         SOURCE STATEMENT

              352+          CSEG
0022 4F4E4553   427+        DB      'ONESEC'
0026 4543       +
0028 0000   E   448+        DW      ONESEC
002A 5800   D   449+        DW      ??0003
002C 1E00       450+        DW      30
002E 82         451+        DB      130
002F 0000       452+        DW      +0
0031 0201   D   453+        DW      TDBASE+(NTASK*20)
0003           454+NTASK    SET     NTASK+1
              455          STD     POTHDI,36,112        ; TERMINAL HANDLER
              456+          EXTRN   POTHDI
              460+          DSEG
0024           461+??0004:  DS      36
              462+          CSEG
0033 52515448   546+        DB      'POTHDI'
0037 4449       +
0039 0000   E   557+        DW      POTHDI
003B 7600   D   558+        DW      ??0004
003D 2400       559+        DW      36
003F 70         560+        DB      112
0040 0000       561+        DW      +0
0042 1601   D   562+        DW      TDBASE+(NTASK*20)
0004           563+NTASK    SET     NTASK+1
              564          STD     POADBG,64,200        ; ACTIVE DEBUGGER
              565+          EXTRN   POADBG
              569+          DSEG
0040           570+??0005:  DS      64
              571+          CSEG
0044 52514144   655+        DB      'POADBG'
0048 4247       +
004A 0000   E   666+        DW      POADBG
004C 9A00   D   667+        DW      ??0005
004E 4000       668+        DW      64
0050 C8         669+        DB      200
0051 0000       670+        DW      +0
0053 2A01   D   671+        DW      TDBASE+(NTASK*20)
0005           672+NTASK    SET     NTASK+1
              673 ;
              674 ;   BUILD INITIAL EXCHANGE TABLE
              675 ;   (USER EXCHANGES CREATED IN TASKS--NOT PRESENT IN IET)
              676 ;
              677          XCHADR POINPX
              678+          EXTRN   POINPX
              680+IET:
0055 0000   E   682+        DW      POINPX
0001           683+NEXCH    SET     NEXCH+1
              684          XCHADR POOUTX
              685+          EXTRN   POOUTX
0057 0000   E   689+        DW      POOUTX
0002           690+NEXCH    SET     NEXCH+1
              691          XCHADR POWAKE
              692+          EXTRN   POWAKE
0059 0000   E   696+        DW      POWAKE
0003           697+NEXCH    SET     NEXCH+1
```

```
ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0        RTCLOK   PAGE    5
RTCLOK:  RMX/80 REAL-TIME CLOCK EXAMPLE


   LOC  OBJ       SEQ        SOURCE STATEMENT


                  698       XCHADR RQDBUG
                  699+        EXTRN    RQDBUG
  005B 0000   E   703+        DW       RQDBUG
  0004             704+NEXCH   SET      NEXCH+1
                  705       XCHADR RQALRM
                  706+        EXTRN    RQALRM
  005D 0000   E   710+        DW       RQALRM
  0005             711+NEXCH   SET      NEXCH+1
                  712       XCHADR RQL6EX
                  713+        EXTRN    RQL6EX
  005F 0000   E   717+        DW       RQL6EX
  0006             718+NEXCH   SET      NEXCH+1
                  719       XCHADR RQL7EX
                  720+        EXTRN    RQL7EX
  0061 0000   E   724+        DW       RQL7EX
  0007             725+NEXCH   SET      NEXCH+1
                  726 ;
                  727 ;   ALLOCATE RAM FOR TASK DESCRIPTORS
                  728 ;
                  729       GENTD
                  730+        DSEG
  0064             731+TDBASE: DS      20*NTASK
                  732+        CSEG
                  733 ;
                  734 ;   BUILD CREATE TABLE
                  735 ;
                  736       CRTAB
                  737+        PUBLIC  RQCRTB
                  738+RQCRTB:
  0063 0000   C   739+        DW       ITT
  0065 05          740+        DB       NTASK
  0066 5500   C   741+        DW       IET
  0068 07          742+        DB       NEXCH
                  743 ;
                  744 ;   END OF RTCLOK CONFIGURATION MODULE
                  745 ;
                  746       END


PUBLIC SYMBOLS
RQCRTB C 0063    RQL0EX D 0000    RQL2EX D 0000    RQL3EX D 0000    RQL4EX D 0000    RQL5EX D 0000

EXTERNAL SYMBOLS
CONSOL E 0000    ONESEC E 0000    RQADBG E 0000    RQALRM E 0000    RQDBUG E 0000    RQINPX E 0000    RQL6EX E 0000
RQL7EX E 0000    RQOUTX E 0000    RQTHDI E 0000    RQWAKE E 0000    UPTIME E 0000

USER SYMBOLS
ADDCHR + 0000    CONSOL E 0000    CRTAB  + 0000    DUMREF D 0000    GENTD  + 0005    IET    C 0055    ITT    C 0000
NEXCH  A 0007    NTASK  A 0005    ONESEC E 0000    RQADBG E 0000    RQALRM E 0000    RQCRTB C 0063    RQDBUG E 0000
RQINPX E 0000    RQL0EX D 0000    RQL2EX D 0000    RQL3EX D 0000    RQL4EX D 0000    RQL5EX D 0000    RQL6EX E 0000
RQL7EX E 0000    RQOUTX E 0000    RQTHDI E 0000    RQWAKE E 0000    STD    + 0000    TDBASE D 00DA    UPTIME E 0000
XCHADR + 0000

ASSEMBLY COMPLETE.  NO ERRORS
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE SCANIN
OBJECT MODULE PLACED IN :F1:SCANIN.OBJ
COMPILER INVOKED BY:  PLM80 :F1:SCANIN.PLM DEBUG
```

```
                 $TITLE('SCANIN:  RMX/80 REAL-TIME CLOCK EXAMPLE')
 1               SCANIN: DO;
                 /*****************************************************************
                 * THIS TYPED PROCEDURE SCANS A STRING OF NUMERIC ASCII CHARACTERS
                 * AND RETURNS THE EQUIVALENT BINARY VALUE.   THE STRING CAN BE A
                 * BINARY, OCTAL, DECIMAL OR HEXADECIMAL REPRESENTATION OF A NUMBER.
                 * THE PARAMETER 'PTR$PTR' CONTAINS THE ADDRESS OF THE BEGINNING
                 * OF THE STRING.  LEADING BLANKS ARE IGNORED.  NOTE THAT SCANIN
                 * ASSUMES THAT THE STRING IT IS SCANNING IS VALID; SURPRISING RESULTS
                 * MAY BE OBTAINED IF IT ISN'T.
                 *****************************************************************/
 2    1          SCAN$INTEGER: PROCEDURE(PTR$PTR) ADDRESS PUBLIC;
 3    2             DECLARE PTR$PTR ADDRESS;
 4    2             DECLARE ORIG$PTR BASED PTR$PTR ADDRESS;
 5    2             DECLARE (SCAN$PTR, UPDATE$PTR, STOP$PTR) ADDRESS;
 6    2             DECLARE CHAR BASED SCAN$PTR BYTE;
 7    2             DECLARE (I, RADIX, INCREMENT) BYTE;
 8    2             DECLARE (NVAL, OVAL) ADDRESS;
 9    2             DECLARE DIGITS(*) BYTE DATA('0123456789ABCDEF');

10    2             SCAN$PTR = ORIG$PTR;
11    2             DO WHILE CHAR = ' ';
12    3                SCAN$PTR = SCAN$PTR + 1;
13    3                END;
14    2             ORIG$PTR = SCAN$PTR;
15    2             DO WHILE (CHAR >= '0' AND CHAR <= '9') OR
                            (CHAR >= 'A' AND CHAR <= 'F');
16    3                SCAN$PTR = SCAN$PTR + 1;
17    3                END;
18    2             UPDATE$PTR = (STOP$PTR := SCAN$PTR) + 1;
19    2             IF CHAR = 'H' THEN DO;
21    3                RADIX = 16;
22    3                END;
23    2             ELSE DO;
24    3                IF CHAR = '0' OR CHAR = '0' THEN DO;
26    4                   RADIX = 8;
27    4                   END;
28    3                ELSE DO;
29    4                   STOP$PTR, SCAN$PTR = (UPDATE$PTR := SCAN$PTR) - 1;
30    4                   IF CHAR = 'B' THEN DO;
32    5                      RADIX = 2;
33    5                      END;
34    4                   ELSE DO;
35    5                      RADIX = 10;
36    5                      IF CHAR <> 'D' THEN DO;
38    6                         STOP$PTR = STOP$PTR + 1;
39    6                         END;
40    5                      END;
41    4                   END;
42    3                END;
```

```
            $EJECT
43   2         SCAN$PTR = ORIG$PTR;
44   2         NVAL, OVAL = 0;
45   2         DO WHILE SCAN$PTR < STOP$PTR;
46   3            DO I = 0 TO LAST(DIGITS);
47   4               IF CHAR = DIGITS(I) THEN INCREMENT = I;
49   4               END;
50   3            IF INCREMENT >= RADIX THEN RETURN 0;
52   3            NVAL = OVAL * RADIX + INCREMENT;
53   3            IF NVAL < OVAL THEN RETURN 0;
55   3            OVAL = NVAL;
56   3            SCAN$PTR = SCAN$PTR + 1;
57   3            END;
58   2         ORIG$PTR = UPDATE$PTR;
59   2         RETURN NVAL;
60   2         END SCAN$INTEGER;
61   1      END;
```

MODULE INFORMATION:

```
    CODE AREA SIZE    = 017AH    378D
    VARIABLE AREA SIZE = 000FH    15D
    MAXIMUM STACK SIZE = 0004H     4D
    67 LINES READ
    0 PROGRAM ERROR(S)
```

END OF PL/M-80 COMPILATION

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE ONESEC
OBJECT MODULE PLACED IN :F1:ONESEC.OBJ
COMPILER INVOKED BY:  PLM80 :F1:ONESEC.PLM DEBUG



           $TITLE('ONESEC:  RMX/80 REAL-TIME CLOCK EXAMPLE')

1          ONESEC: DO;
           /**********************************************************************
           * PRIORITY:  130
           * RECEIVES:     MESSAGE-------FROM-------VIA---------COMMENTS
           *               TIME$OUT$TYPE   RMX/80    WAITX    1 SECOND TIME OUT
           *
           * SENDS:        MESSAGE--------TO--------VIA---------COMMENTS
           *               ADVANCE$TIME  UPTIME    UPTIMX    ADVANCE CLOCK 1 SECOND
           *
           * USES PUBLIC PROCS:  RQWAIT,RQSEND,RQCXCH
           * READS PUBLIC DATA:  NONE
           * WRITES PUBLIC DATA:  NONE
           *
           * FUNCTION:  THIS IS THE HIGHEST PRIORITY TASK IN THE SYSTEM.   IT
           *   SIMPLY WAITS FOR ONE SECOND AND THEN INFORMS UPTIME THAT
           *   ONE SECOND HAS PASSED.
           *
           *   NOTE THAT IN ORDER TO MAINTAIN SYNCHRONIZATION WITH THE
           *   HARDWARE CLOCK, ONESEC MUST DO ITS THING AND GET BACK ON
           *   THE WAIT LIST (ISSUE AN RQWAIT) BEFORE THE NEXT "TICK" OF
           *   THE CLOCK (50 MS).
           **********************************************************************/
```

```
                    $EJECT
                    $INCLUDE (:F1:EXCH.ELT)
   2  1  =   DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
         =     MESSAGE$HEAD ADDRESS,
         =     MESSAGE$TAIL ADDRESS,
         =     TASK$HEAD ADDRESS,
         =     TASK$TAIL ADDRESS,
         =     EXCHANGE$LINK ADDRESS)';
                    $INCLUDE (:F1:MSG.ELT)
   3  1  =   DECLARE MSG$HDR LITERALLY '
         =     LINK ADDRESS,
         =     LENGTH ADDRESS,
         =     TYPE BYTE,
         =     HOME$EXCHANGE ADDRESS,
         =     RESPONSE$EXCHANGE ADDRESS';
         =
   4  1  =   DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
         =     MSG$HDR,
         =     REMAINDER(1) BYTE)';

            /*MISCELLANEOUS VARIABLES*/
   5  1      DECLARE WAIT$PTR  ADDRESS;
   6  1      DECLARE FOREVER   LITERALLY 'WHILE 1';

            /*MESSAGES*/
   7  1      DECLARE ADVANCE$TIME  STRUCTURE(
                         MSG$HDR) PUBLIC;

            /*EXCHANGES*/
   8  1      DECLARE WAITX EXCHANGE$DESCRIPTOR;
   9  1      DECLARE UPTIMX EXCHANGE$DESCRIPTOR PUBLIC;

            /*EXTERNAL PROCEDURES*/
  10  1      RQCXCH:  PROCEDURE(EXCHANGE$PTR) EXTERNAL;
  11  2        DECLARE EXCHANGE$PTR ADDRESS;
  12  2        END RQCXCH;

  13  1      RQSEND:  PROCEDURE(EXCHANGE$PTR,MESSAGE$PTR) EXTERNAL;
  14  2        DECLARE (EXCHANGE$PTR,MESSAGE$PTR) ADDRESS;
  15  2        END RQSEND;

  16  1      RQWAIT:  PROCEDURE(EXCHANGE$PTR,DELAY) ADDRESS EXTERNAL;
  17  2        DECLARE (EXCHANGE$PTR,DELAY) ADDRESS;
  18  2        END RQWAIT;
```

```
              $EJECT
              /*INITIALIZATION*/
  19   1      ONESEC: PROCEDURE PUBLIC;
  20   2         CALL RQCXCH(.WAITX);
  21   2         CALL RQCXCH(.UPTIMX);
  22   2         ADVANCE$TIME.LENGTH = SIZE(ADVANCE$TIME);
  23   2         ADVANCE$TIME.TYPE = 100;

              /*MAIN PROCESSING LOOP*/
  24   2         DO FOREVER;
  25   3            WAIT$PTR = RQWAIT(.WAITX,20);        /*ONE SECOND*/
  26   3            CALL RQSEND(.UPTIMX,.ADVANCE$TIME);  /*TELL UPTIME TO BUMP CLOCK*/
  27   3            END;
  28   2         END ONESEC;
  29   1      END;


MODULE INFORMATION:

     CODE AREA SIZE    = 0030H     48D
     VARIABLE AREA SIZE = 001FH    31D
     MAXIMUM STACK SIZE = 0002H     2D
     84 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE CONSOL
OBJECT MODULE PLACED IN :F1:CONSOL.OBJ
COMPILER INVOKED BY:  PLM80 :F1:CONSOL.PLM DEBUG


            $TITLE('CONSOL.  RMX/80 REAL-TIME CLOCK EXAMPLE')
    1       CONSOL: DO;
            /*****************************************************************
            * PRIORITY: 180
            * RECEIVES: MESSAGE------FROM--------VIA----------COMMENTS
            *           COMAND      T.H.      CREADX    T.H. RESPONSE
            *           REPLY       T.H.      CWRITX    T.H. RESPONSE
            *
            * SENDS:    MESSAGE-------TO-------VIA---------COMMENTS
            *           COMAND      T.H.      RQINPX    REQUEST INPUT
            *           REPLY       T.H.      RQOUTX    RESPONSE TO COMMAND
            *           SET$TIME    UPTIME    UPTIMX    SET REAL/ALARM TIME
            *           ENABLE$DISPLAY UPTIME  UPTIMX   START/STOP TIME DISPLAY
            *
            * USES PUBLIC PROCS: RQWAIT, RQSEND, RQCXCH, SCAN$INTEGER
            * READS PUBLIC DATA: NONE
            * WRITES PUBLIC DATA: NONE
            *
            * FUNCTION: THIS TASK COORDINATES INTERACTION WITH THE USER AT THE
            *  TERMINAL.  IT:
            *     1. SIGNS ON WHEN THE SYSTEM COMES UP.
            *     2. ACCEPTS COMMANDS.
            *     3. EXECUTES VALID COMMANDS, WRITES AN ERROR MESSAGE
            *        TO THE DISPLAY IF IT RECEIVES AN INVALID COMMAND.
            *     4. DISABLES THE AUTOMATIC TIME DISPLAY DURING DATA ENTRY.
            *  VALID COMMANDS ARE.
            *     C  (SET REAL-TIME CLOCK)
            *     A  (SET ALARM)
            *     ?  (DISPLAY COMMAND MENU)
            *        (A "NULL" COMMAND -CR ONLY- STOPS THE TIME DISPLAY
            *  WHEN T OR A IS ENTERED, CONSOL PROMPTS FOR THE TIME (HH:MM:SS?)
            *****************************************************************/
```

```
            $EJECT
            $INCLUDE (:F1:EXCH.ELT)
  2   1 =   DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
      =       MESSAGE$HEAD ADDRESS,
      =       MESSAGE$TAIL ADDRESS,
      =       TASK$HEAD ADDRESS,
      =       TASK$TAIL ADDRESS,
      =       EXCHANGE$LINK ADDRESS)';
            $INCLUDE (:F1:MSG.ELT)
  3   1 =   DECLARE MSG$HDR LITERALLY '
      =       LINK ADDRESS,
      =       LENGTH ADDRESS,
      =       TYPE BYTE,
      =       HOME$EXCHANGE ADDRESS,
      =       RESPONSE$EXCHANGE ADDRESS',
      =
  4   1 =   DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
      =       MSG$HDR,
      =       REMAINDER(1) BYTE)';
```

```
               .      $EJECT
                      /*EXCHANGES*/
      5    1     DECLARE (RQOUTX,
                         RQINPX,
                         UPTIMX)  EXCHANGE$DESCRIPTOR EXTERNAL;
      6    1     DECLARE  (CREADX,CWRITX) EXCHANGE$DESCRIPTOR;

                      /*MISCELLANEOUS VARIABLES*/
      7    1     DECLARE  FOREVER       LITERALLY 'WHILE 1';
      8    1     DECLARE  WAIT$PTR       ADDRESS;
      9    1     DECLARE  WRITE$TYPE     LITERALLY '12';
     10    1     DECLARE  READ$TYPE      LITERALLY '8';
     11    1     DECLARE  READ$PTR       ADDRESS;
     12    1     DECLARE  CR             LITERALLY '0DH';  /*CARRIAGE RETURN*/
     13    1     DECLARE  LF             LITERALLY '0AH';  /*LINE FEED*/
     14    1     DECLARE  EXPECTING$TIME BYTE; /*NEXT COMMAND SHOULD BE TIME*/
     15    1     DECLARE  TRUE           LITERALLY '0FFH';
     16    1     DECLARE  FALSE          LITERALLY '000H';
```

```
              $EJECT

              /*MESSAGES*/
   17   1     DECLARE  (REPLY,COMAND) STRUCTURE (
                       MSG$HDR,
                       STATUS      ADDRESS,
                       BUFFER$ADR  ADDRESS,
                       COUNT       ADDRESS,
                       ACTUAL      ADDRESS);
   18   1      DECLARE  ENABLE$DISPLAY  STRUCTURE (
                       MSG$HDR)    EXTERNAL;
   19   1      DECLARE  SET$TIME        STRUCTURE (
                       MSG$HDR,
                       NEW$HR      BYTE,
                       NEW$MIN     BYTE,
                       NEW$SEC     BYTE) EXTERNAL;

   20   1     DECLARE  SIGNON$MSG(*)  BYTE DATA(
                       CR,LF,'RTCLOK:  VERSION 1.0',CR,LF,LF,
                       'ALL TIMES ARE 24-HOUR FORMAT: HH.MM:SS',CR,LF,
                       '    09:22:30 = 9.22:30 AM',CR,LF,
                       '    17:30:00 = 5:30 PM',CR,LF,
                       '    00.00.00 = MIDNIGHT',CR,LF,
                       '    24.00:00 = NO TIME - CLOCK IS OFF',CR,LF);
   21   1     DECLARE  MENU$MSG(*)  BYTE DATA(
                       CR,LF,
                       'AVAILABLE COMMANDS:',CR,LF,
                       '    C = SET REAL-TIME CLOCK',CR,LF,
                       '    A = SET ALARM',CR,LF,
                       '    ? = DISPLAY THIS COMMAND MENU',CR,LF,
                       'ENTERING A "NULL COMMAND" (CARRIAGE RETURN ONLY)',CR,LF,
                       'SUSPENDS THE AUTOMATIC TIME DISPLAY.',CR,LF,
                       'IF YOU ENTER C OR A, THE SYSTEM WILL PROMPT YOU TO',
                       CR,LF,'ENTER A TIME.  ',
                       'REMEMBER TO PUT IN COLONS.',CR,LF,LF,
                       'TO SHUT OFF THE ALARM, SET IT TO 24:00:00.',CR,LF);
   22   1     DECLARE  TIME$MSG(*)  BYTE DATA(
                       CR,LF,'HH.MM:SS?',CR,LF);
   23   1     DECLARE  INVALID$MSG(*)  BYTE DATA(
                       CR,LF,'INVALID COMMAND',CR,LF);
   24   1     DECLARE  BAD$TIME$MSG (*)  BYTE DATA(
                       CR,LF,'TIME IS INVALID',CR,LF);
   25   1     DECLARE  COMAND$MSG(80)  BYTE;
```

```
              $EJECT
              /*EXTERNAL PROCEDURES*/
   26   1     ROSEND:  PROCEDURE(EXCHANGE$PTR,MESSAGE$PTR) EXTERNAL;
   27   2       DECLARE (EXCHANGE$PTR,MESSAGE$PTR) ADDRESS;
   28   2       END ROSEND;
   29   1     RQWAIT:  PROCEDURE(EXCHANGE$PTR,DELAY) ADDRESS EXTERNAL;
   30   2       DECLARE (EXCHANGE$PTR,DELAY) ADDRESS;
   31   2       END RQWAIT;
   32   1     RQCXCH:  PROCEDURE(EXCHANGE$PTR) EXTERNAL;
   33   2       DECLARE EXCHANGE$PTR ADDRESS;
   34   2       END RQCXCH;
   35   1     SCAN$INTEGER: PROCEDURE(PTR) ADDRESS EXTERNAL;
   36   2       DECLARE PTR ADDRESS;
   37   2       END SCAN$INTEGER;
```

```
                  $EJECT
                  /*INTERNAL PROCEDURES*/
     38    1       TIME$ON:  PROCEDURE;
                      /*TURN ON AUTO TIME DISPLAY*/
     39    2         ENABLE$DISPLAY.LENGTH = SIZE(ENABLE$DISPLAY);
     40    2         ENABLE$DISPLAY.TYPE = 140;   /*ENABLE*/
     41    2         CALL RQSEND(.UPTIMX, .ENABLE$DISPLAY);
     42    2         END TIME$ON;

     43    1       TIME$OFF:  PROCEDURE;
                      /*SHUT OFF AUTO TIME DISPLAY*/
     44    2         ENABLE$DISPLAY.LENGTH = SIZE(ENABLE$DISPLAY);
     45    2         ENABLE$DISPLAY.TYPE = 130;   /*DISABLE*/
     46    2         CALL RQSEND(.UPTIMX, .ENABLE$DISPLAY);
     47    2         END TIME$OFF;

     48    1       ASK$FOR$TIME:  PROCEDURE;
                      /*DISPLAY "HH:MM:SS?"*/
     49    2         REPLY.BUFFER$ADR = .TIME$MSG;
     50    2         REPLY.COUNT = SIZE(TIME$MSG);
     51    2         CALL RQSEND(.ROOUTX, .REPLY);
     52    2         WAIT$PTR = RQWAIT(.CWRITX, 0);   /*WAIT FOR T.H. TO FINISH*/
     53    2         EXPECTING$TIME = TRUE;   /*NEXT COMMAND SHOULD BE TIME SETTING*/
     54    2         END ASK$FOR$TIME;

     55    1       INVALID:  PROCEDURE;
                      /*DISPLAY ERROR MESSAGE*/
     56    2         CALL TIME$OFF;  /*SHUT OFF AUTO TIME DISPLAY*/
     57    2         REPLY.BUFFER$ADR = .INVALID$MSG;
     58    2         REPLY.COUNT = SIZE(INVALID$MSG);
     59    2         CALL RQSEND(.ROOUTX, .REPLY);
     60    2         WAIT$PTR = RQWAIT(.CWRITX, 0);   /*WAIT FOR T.H. TO FINISH*/
     61    2         CALL TIME$ON;   /*TURN AUTO TIME DISPLAY BACK ON*/
     62    2         END INVALID;

     63    1       BAD$TIME:  PROCEDURE;
                      /*REJECT INVALID TIME SETTING*/
     64    2         CALL TIME$OFF;  /*SHUT OFF AUTO TIME DISPLAY*/
     65    2         REPLY.BUFFER$ADR = .BAD$TIME$MSG;
     66    2         REPLY.COUNT = SIZE(BAD$TIME$MSG);
     67    2         CALL RQSEND(.ROOUTX, .REPLY);
     68    2         WAIT$PTR = RQWAIT(.CWRITX, 0);   /*WAIT FOR T.H. TO FINISH*/
     69    2         CALL TIME$ON;   /*RESUME AUTO TIME DISPLAY*/
     70    2         END BAD$TIME;
```

```
                    $EJECT
   71   1    SET$REAL: PROCEDURE;
                    /*SET REAL-TIME CLOCK*/
   72   2        CALL TIME$OFF;  /*TURN OFF AUTO TIME DISPLAY*/
   73   2        SET$TIME.LENGTH = SIZE(SET$TIME);
   74   2        SET$TIME.TYPE = 120;   /*SET REAL-TIME CLOCK*/
   75   2        CALL ASK$FOR$TIME;  /*GET THE TIME VALUE TO SET*/
   76   2        END SET$REAL;

   77   1    SET$ALARM: PROCEDURE;
                    /*SET ALARM*/
   78   2        CALL TIME$OFF;  /*TURN OFF AUTO TIME DISPLAY*/
   79   2        SET$TIME.LENGTH = SIZE(SET$TIME);
   80   2        SET$TIME.TYPE = 110;   /*SET ALARM CLOCK*/
   81   2        CALL ASK$FOR$TIME;  /*GET THE TIME VALUE TO SET*/
   82   2        END SET$ALARM;

   83   1    MENU: PROCEDURE;
                    /*DISPLAY VALID COMMANDS*/
   84   2        CALL TIME$OFF;  /*SHUT OFF AUTO TIME DISPLAY*/
   85   2        REPLY.BUFFER$ADR = .MENU$MSG;
   86   2        REPLY.COUNT = SIZE(MENU$MSG);
   87   2        CALL RQSEND(.RQOUTX,.REPLY);
   88   2        WAIT$PTR = RQWAIT(.CWRITX,0);  /*WAIT FOR T.H. TO FINISH*/
   89   2        CALL TIME$ON;  /*RESUME AUTO TIME DISPLAY*/
   90   2        END MENU;

   91   1    SUPPLY$TIME: PROCEDURE;
                    /*PASS THE TIME SETTING TO UPTIME*/
   92   2        DECLARE PTR    ADDRESS;
   93   2        PTR = COMAND.BUFFER$ADR;
   94   2        SET$TIME.NEW$HR = SCAN$INTEGER(.PTR);    /*ASCII TO BINARY*/
   95   2        PTR = PTR + 1;    /*GO PAST COLON*/
   96   2        SET$TIME.NEW$MIN = SCAN$INTEGER(.PTR);
   97   2        PTR = PTR + 1;    /*GO PAST COLON*/
   98   2        SET$TIME.NEW$SEC = SCAN$INTEGER(.PTR);
                    /*VALIDATE THE SETTING*/
   99   2        IF (SET$TIME.NEW$HR < 24 AND
                        SET$TIME.NEW$MIN < 60 AND
                        SET$TIME.NEW$SEC < 60)
                      OR
                       (SET$TIME.NEW$HR = 24 AND
                        SET$TIME.NEW$MIN = 0 AND
                        SET$TIME.NEW$SEC = 0)
                    THEN CALL RQSEND(.UPTIMX,.SET$TIME);
  101   2        ELSE CALL BAD$TIME;  /*DISPLAY ERROR MESSAGE*/
  102   2        CALL TIME$ON;  /*RESUME AUTO TIME DISPLAY*/
  103   2        EXPECTING$TIME = FALSE;  /*NEXT COMMAND SHOULD BE A,T,?*/
  104   2        END SUPPLY$TIME;

  105   1    REJECT$COMMAND: PROCEDURE;
                    /*THROW OUT AN INVALID COMMAND*/
  106   2        CALL INVALID;  /*DISPLAY ERROR MESSAGE*/
  107   2        CALL MENU;  /*DISPLAY VALID COMMANDS*/
  108   2        END REJECT$COMMAND;
```

```
                $EJECT
                /*INITIALIZATION*/
109    1        CONSOL:  PROCEDURE PUBLIC;
110    2           CALL RQCXCH(.CWRITX);   /*WHERE WE WAIT FOR T.H. OUTPUT TO COMPLETE*/
111    2           REPLY.LENGTH = SIZE(REPLY);
112    2           REPLY.TYPE = WRITE$TYPE;
113    2           REPLY.RESPONSE$EXCHANGE = .CWRITX;
114    2           REPLY.BUFFER$ADR = .SIGNON$MSG;
115    2           REPLY.COUNT = SIZE(SIGNON$MSG);
116    2           CALL RQSEND(.RQOUTX,.REPLY);   /*DISPLAY SIGNON MESSAGE*/
117    2           WAIT$PTR = RQWAIT(.CWRITX,0);
118    2           CALL MENU;   /*DISPLAY VALID COMMANDS*/
119    2           CALL RQCXCH(.CREADX);   /*WHERE WE WAIT FOR T.H. INPUT*/
120    2           COMAND.LENGTH = SIZE(COMAND);
121    2           COMAND.TYPE = READ$TYPE;
122    2           COMAND.RESPONSE$EXCHANGE = .CREADX;
123    2           COMAND.BUFFER$ADR = .COMAND$MSG;
124    2           COMAND.COUNT = SIZE(COMAND$MSG);
125    2           EXPECTING$TIME = FALSE;

                /*MAIN PROCESSING LOOP*/
126    2           DO FOREVER;
                   /*REQUEST A COMMAND FROM THE TERMINAL HANDLER*/
127    3              CALL RQSEND(.RQINPX,.COMAND);
128    3              READ$PTR = RQWAIT(.CREADX,0);   /*WAIT FOR T.H. TO FINISH*/

                   /*CALL A PROCEDURE TO PROCESS THE COMMAND*/
129    3              IF COMAND$MSG(0) = 'C'
                         THEN CALL SET$REAL;   /*SET THE CLOCK*/
131    3              ELSE IF COMAND$MSG(0) = 'A'
                         THEN CALL SET$ALARM;   /*SET THE ALARM*/
133    3              ELSE IF COMAND$MSG(0) = '?'
                         THEN CALL MENU;   /*DISPLAY VALID COMMANDS*/
135    3              ELSE IF COMAND$MSG(0) = CR   /*CARRIAGE RETURN ONLY*/
                         THEN CALL TIME$OFF;   /*SHUT OFF AUTO TIME DISPLAY*/
137    3              ELSE IF EXPECTING$TIME
                         THEN CALL SUPPLY$TIME;   /*GET CLOCK/ALARM SETTING VALUE*/
139    3              ELSE CALL REJECT$COMMAND;   /*REJECT THE INVALID COMMAND*/
140    3              END;
141    2        END CONSOL;
142    1        END;


MODULE INFORMATION:

    CODE AREA SIZE     = 047CH   1148D
    VARIABLE AREA SIZE = 008DH    141D
    MAXIMUM STACK SIZE = 0008H      8D
    266 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

PL/M-80 COMPILER   UPTIME: RMX/80 REAL-TIME CLOCK EXAMPLE   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE UPTIME
OBJECT MODULE PLACED IN :F1:UPTIME.OBJ
COMPILER INVOKED BY: PLM80 :F1:UPTIME.PLM DEBUG


```
              $TITLE('UPTIME:  RMX/80 REAL-TIME CLOCK EXAMPLE')

1             UPTIME: DO;
              /************************************************************
              * PRIORITY: 160
              * RECEIVES:       MESSAGE-------FROM-------VIA-------COMMENTS
              *              ADVANCE$TIME    ONESEC    UPTIMX   ADVANCE CLOCK 1 SECOND
              *                SET$TIME      CONSOL    UPTIMX   SET REAL OR ALARM
              *              ENABLE$DISPLAY  CONSOL    UPTIMX   AUTO TIME ON/OFF
              *                SHOW$TIME   TERM.HNDLR  UWRITX   T.H. RESPONSE
              *
              * SENDS:          MESSAGE--------TO--------VIA------COMMENTS
              *                SHOW$TIME   TERM.HNDLR   RQOUTX   AUTO TIME DISPLAY
              *
              * USES PUBLIC PROCS:  RQWAIT,RQSEND,RQCXCH,NUMOUT
              * READS PUBLIC DATA:  REAL$TIME,ALARM$TIME
              * WRITES PUBLIC DATA: PEAL$TIME,ALARM$TIME
              *
              * FUNCTION:  THIS TASK IS IN CHARGE OF MAINTAINING THE REAL-TIME
              *   CLOCK AND THE ALARM.  THIS INCLUDES:
              *      1. SET REAL-TIME CLOCK.
              *      2. SET ALARM.
              *      3. ADVANCE REAL-TIME CLOCK 1 SECOND (IF IT IS RUNNING).
              *      4. DISPLAY THE VALUES OF REAL-TIME CLOCK AND ALARM
              *         (IF DISPLAY IS ENABLED).
              *      5. IF REAL-TIME >= ALARM SETTING, ADD ALARM INDICATION
              *         TO THE TIME DISPLAY.
              ************************************************************/
```

```
PL/M-80 COMPILER    UPTIME: RMX/80 REAL-TIME CLOCK EXAMPLE                              PAGE  2


                 $EJECT
                 $INCLUDE (:F1:EXCH.ELT)
  2  1  =    DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
     =         MESSAGE$HEAD ADDRESS,
     =         MESSAGE$TAIL ADDRESS,
     =         TASK$HEAD ADDRESS,
     =         TASK$TAIL ADDRESS,
     =         EXCHANGE$LINK ADDRESS)';
                 $INCLUDE (:F1:MSG.ELT)
  3  1  =    DECLARE MSG$HDR LITERALLY '
     =         LINK ADDRESS,
     =         LENGTH ADDRESS,
     =         TYPE BYTE,
     =         HOME$EXCHANGE ADDRESS,
     =         RESPONSE$EXCHANGE ADDRESS';
     =
  4  1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
     =         MSG$HDR,
     =         REMAINDER(1) BYTE)';

                 /*MISCELLANEOUS VARIABLES*/
  5  1      DECLARE FOREVER          LITERALLY 'WHILE 1';
  6  1      DECLARE WRITE$TYPE       LITERALLY '12';
  7  1      DECLARE TRUE             LITERALLY '0FFH',
  8  1      DECLARE FALSE            LITERALLY '000H';
  9  1      DECLARE (DISPLAY$ENABLED, ALARM$ENABLED)   BYTE;
 10  1      DECLARE TH$RTN$MSG       ADDRESS;
 11  1      DECLARE IN$MSG$ADDR      ADDRESS;
 12  1      DECLARE DECODE$MSG       BASED IN$MSG$ADDR STRUCTURE (
                    MSG$HDR);
 13  1      DECLARE FIRST$TIME       BYTE;
 14  1      DECLARE CR               LITERALLY '0DH';  /*CARRIAGE RETURN*/
 15  1      DECLARE LF               LITERALLY '0AH';  /*LINE FEED*/

                 /*EXCHANGES*/
 16  1      DECLARE (UPTIMX,ROOUTX) EXCHANGE$DESCRIPTOR EXTERNAL;
 17  1      DECLARE UWRITX EXCHANGE$DESCRIPTOR;

                 /*INPUT MESSAGES*/
 18  1      DECLARE ADVANCE$TIME STRUCTURE (
                       MSG$HDR) EXTERNAL;
 19  1      DECLARE ENABLE$DISPLAY STRUCTURE (
                       MSG$HDR) PUBLIC;
 20  1      DECLARE SET$TIME STRUCTURE (
                       MSG$HDR,
                       NEW$HR       BYTE,
                       NEW$MIN      BYTE,
                       NEW$SEC      BYTE) PUBLIC;

                 /*PUBLIC TIME VARIABLES*/
 21  1      DECLARE (REAL$TIME, ALARM$TIME) STRUCTURE (
                       HR    BYTE,
                       MIN   BYTE,
                       SEC   BYTE) PUBLIC;
```

```
                    $EJECT
                    /*EXTERNAL PROCEDURES*/
22   1    RQCXCH:  PROCEDURE(EXCHANGE$PTR) EXTERNAL;
23   2       DECLARE EXCHANGE$PTR ADDRESS;
24   2       END RQCXCH;
25   1    RQSEND:  PROCEDURE(EXCHANGE$PTR,MESSAGE$PTR) EXTERNAL;
26   2       DECLARE (EXCHANGE$PTR,MESSAGE$PTR) ADDRESS;
27   2       END RQSEND;
28   1    RQWAIT:  PROCEDURE(EXCHANGE$PTR,DELAY) ADDRESS EXTERNAL;
29   2       DECLARE (EXCHANGE$PTR,DELAY) ADDRESS;
30   2       END RQWAIT;
31   1    NUMOUT:  PROCEDURE(VALUE,BASE,LC,BUFADR,WIDTH) EXTERNAL;
32   2       DECLARE (VALUE,BUFADR) ADDRESS;
33   2       DECLARE (BASE,LC,WIDTH) BYTE;
34   2       END NUMOUT;
```

```
                     $EJECT
                     /*INTERNAL PROCEDURES*/
35    1      ADVANCE:  PROCEDURE;
                        /*BUMP THE CLOCK IF IT'S RUNNING*/
36    2          IF REAL$TIME.HR <> 24 THEN DO;
                        /*CLOCK IS RUNNING, ADVANCE IT*/
38    3              REAL$TIME.SEC = REAL$TIME.SEC + 1;
39    3              IF REAL$TIME.SEC = 60 THEN DO;
41    4                  REAL$TIME.SEC = 0;
42    4                  REAL$TIME.MIN = REAL$TIME.MIN + 1;
43    4                  IF REAL$TIME.MIN = 60 THEN DO;
45    5                      REAL$TIME.MIN = 0;
46    5                      REAL$TIME.HR = REAL$TIME.HR + 1;
47    5                      IF REAL$TIME.HR = 24 THEN DO;
                                /*ROLL TO NEXT DAY*/
49    6                          REAL$TIME.HR = 0;
50    6                          REAL$TIME.MIN = 0;
51    6                          REAL$TIME.SEC = 0;
52    6                          END;
53    5                      END;
54    4                  END;
55    3              END;
56    2          END ADVANCE;

57    1      SET:  PROCEDURE;
                    /*SET THE CLOCK OR ALARM ACCORDING TO MSG TYPE*/
58    2          IF SET$TIME.TYPE = 120 THEN DO;
60    3              REAL$TIME.HR = SET$TIME.NEW$HR;
61    3              REAL$TIME.MIN = SET$TIME.NEW$MIN;
62    3              REAL$TIME.SEC = SET$TIME.NEW$SEC;
63    3              END;
64    2          ELSE IF SET$TIME.TYPE = 110 THEN DO;
66    3              ALARM$TIME.HR = SET$TIME.NEW$HR;
67    3              ALARM$TIME.MIN = SET$TIME.NEW$MIN;
68    3              ALARM$TIME.SEC = SET$TIME.NEW$SEC;
69    3              IF ALARM$TIME.HR = 24 THEN    /*24 MEANS ALARM IS NOT SET*/
70    3                  ALARM$ENABLED = FALSE;
71    3              ELSE ALARM$ENABLED = TRUE;
72    3              END;
             END SET;

74    1      ENAB$DISPLAY:  PROCEDURE;
                    /*TURN AUTO TIME DISPLAY ON/OFF*/
75    2          IF ENABLE$DISPLAY.TYPE = 140 THEN
76    2              DISPLAY$ENABLED = TRUE;
77    2          ELSE IF ENABLE$DISPLAY.TYPE = 130 THEN
78    2              DISPLAY$ENABLED = FALSE;
             END ENAB$DISPLAY;
```

PL/M-80 COMPILER   UPTIME: RMX/80 REAL-TIME CLOCK EXAMPLE   PAGE   5

```
              $EJECT
  80   1      SHOW:  PROCEDURE;
              /*DISPLAY SETTINGS OF CLOCK AND ALARM*/
  81   2          DECLARE SHOW$TIME    STRUCTURE(
                          MSG$HDR,
                          STATUS       ADDRESS,
                          BUFFER$ADR ADDRESS,
                          COUNT        ADDRESS,
                          ACTUAL       ADDRESS);
  82   2          DECLARE TIME$MSG    STRUCTURE(
                          TIME$LABL(9)    BYTE,
                          T$HR(2)         BYTE,
                          COLON1          BYTE,
                          T$MIN(2)        BYTE,
                          COLON2          BYTE,
                          T$SEC(2)        BYTE,
                          ALARM$LABL(8)   BYTE,
                          A$HR(2)         BYTE,
                          COLON3          BYTE,
                          A$MIN(2)        BYTE,
                          COLON4          BYTE,
                          A$SEC(2)        BYTE,
                          CARRIAGE        BYTE,
                          ALARM$MSG(5)    BYTE).
  83   2      DECLARE BELL LITERALLY '07H';
  84   2      DECLARE ALARM$INDICATION(5) BYTE DATA('!',BELL,'!',BELL,CR);
  85   2      DECLARE TIME$CONST(9) BYTE DATA('    TIME=');
  86   2      DECLARE ALARM$CONST(8) BYTE DATA('--ALARM=');


  87   2          IF FIRST$TIME THEN DO;
                  /*INITIALIZE MESSAGE TEXT*/
  89   3          CALL MOVE(SIZE(TIME$MSG.TIME$LABL),.TIME$CONST,.TIME$MSG.TIME$LABL);
  90   3          TIME$MSG.COLON1, TIME$MSG.COLON2,
                  TIME$MSG.COLON3, TIME$MSG.COLON4 = ':';
  91   3          CALL MOVE(SIZE(TIME$MSG.ALARM$LABL),.ALARM$CONST,.TIME$MSG.ALARM$LABL);
  92   3          CALL MOVE(SIZE(TIME$MSG.ALARM$MSG),
                      .ALARM$INDICATION,.TIME$MSG.ALARM$MSG);
                  /*INITIALIZE MESSAGE FIELDS*/
  93   3          SHOW$TIME.LENGTH = SIZE(SHOW$TIME);
  94   3          SHOW$TIME.TYPE = WRITE$TYPE,
  95   3          SHOW$TIME.RESPONSE$EXCHANGE = .UWRITX.
  96   3          SHOW$TIME.BUFFER$ADR = .TIME$MSG,
  97   3          END.
```

```
                    $EJECT
                    /*WAIT FOR T.H. TO OUTPUT PREVIOUS MESSAGE*/
  98   2            IF NOT FIRST$TIME
                        THEN TH$RTN$MSG = RQWAIT(.UWRIT%,0);

                    /*CONVERT TIME & ALARM TO ASCII*/
 100   2            CALL NUMOUT(REAL$TIME.HR,10,'0',.TIME$MSG.T$HR,2);
 101   2            CALL NUMOUT(REAL$TIME.MIN,10,'0',.TIME$MSG.T$MIN,2);
 102   2            CALL NUMOUT(REAL$TIME.SEC,10,'0',.TIME$MSG.T$SEC,2);
 103   2            CALL NUMOUT(ALARM$TIME.HR,10,'0',.TIME$MSG.A$HR,2);
 104   2            CALL NUMOUT(ALARM$TIME.MIN,10,'0',.TIME$MSG.A$MIN,2);
 105   2            CALL NUMOUT(ALARM$TIME.SEC,10,'0',.TIME$MSG.A$SEC,2);

                    /*CHECK FOR ALARM INDICATION*/
                    /*ASSUME NO ALARM - SET COUNT TO EXCLUDE ALARM INDICATION*/
 106   2            SHOW$TIME.COUNT = SIZE(TIME$MSG) -
                                      SIZE(TIME$MSG.ALARM$MSG);
 107   2            TIME$MSG.CARRIAGE = CR;
 108   2            IF ALARM$ENABLED
                        THEN IF REAL$TIME.HR <> 24   /*24=CLOCK NOT RUNNING*/
                            THEN IF (REAL$TIME.HR = ALARM$TIME.HR AND
                                     REAL$TIME.MIN = ALARM$TIME.MIN AND
                                     REAL$TIME.SEC >= ALARM$TIME.SEC)
                                 OR (REAL$TIME.HR = ALARM$TIME.HR AND
                                     REAL$TIME.MIN > ALARM$TIME.MIN)
                                 OR (REAL$TIME.HR > ALARM$TIME.HR)
                                 THEN DO;  /*BUMP COUNT TO INCLUDE ALARM INDICATION*/
 112   3                             SHOW$TIME.COUNT = SIZE(TIME$MSG);
 113   3                             TIME$MSG.CARRIAGE = '!';  /*ZAP 1ST CR IN MSG*/
 114   3                             END;

                    /*SEND THE MESSAGE TO THE TERMINAL HANDLER*/
 115   2            CALL RQSEND(.RQOUT%,.SHOW$TIME);
 116   2            FIRST$TIME = FALSE;
 117   2            END SHOW;
```

```
                $EJECT
118    1        UPTIME:  PROCEDURE PUBLIC;
                /*INITIALIZATION*/
119    2        CALL ROCXCH(,UWRITX);   /*WHERE WE WAIT FOR T.H. TO COMPLETE*/
120    2        REAL$TIME.HR,ALARM$TIME.HR = 24;
121    2        REAL$TIME.MIN,REAL$TIME.SEC = 0;
122    2        ALARM$TIME.MIN,ALARM$TIME.SEC = 0;
123    2        DISPLAY$ENABLED,ALARM$ENABLED = FALSE;
124    2        FIRST$TIME = TRUE;

                /*MAIN PROCESSING LOOP*/
125    2        DO FOREVER;
                    /*GET THE NEXT MESSAGE, IF THERE IS ONE, OTHERWISE WAIT*/
126    3            IN$MSG$ADDR = ROWAIT(,UPTIMX,0);

                    /*CALL A ROUTINE TO PROCESS THE MESSAGE*/
127    3            IF DECODE$MSG.TYPE = 100
                        THEN CALL ADVANCE;   /*BUMP THE CLOCK 1 SECOND*/
129    3            ELSE IF (DECODE$MSG.TYPE = 110 OR
                            DECODE$MSG.TYPE = 120)
                        THEN CALL SET;   /*SET THE CLOCK OR THE ALARM*/
131    3            ELSE IF (DECODE$MSG.TYPE = 130 OR
                            DECODE$MSG.TYPE = 140)
                        THEN CALL ENAB$DISPLAY;   /*TURN THE AUTO TIME DISPLAY ON/OFF*/

                    /*CHECK FOR TIME DISPLAY AND ALARM*/
                    IF DISPLAY$ENABLED
                        THEN IF DECODE$MSG.TYPE = 100
                            THEN CALL SHOW;   /*DISPLAY TIME & ALARM SETTINGS*/
136    3            END;
137    2        END UPTIME;
138    1        END;
```

MODULE INFORMATION:

```
    CODE AREA SIZE    = 02DAH    730D
    VARIABLE AREA SIZE = 0064H    100D
    MAXIMUM STACK SIZE = 000AH     10D
    258 LINES READ
    0 PROGRAM ERROR(S)
```

END OF PL/M-80 COMPILATION

PL/M-80 COMPILER   NUMOUT: RMX/80 REAL-TIME CLOCK EXAMPLE   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE NUMOUT
OBJECT MODULE PLACED IN :F1:NUMOUT.OBJ
COMPILER INVOKED BY: PLM80 :F1:NUMOUT.PLM DEBUG


```
              $TITLE('NUMOUT:  RMX/80 REAL-TIME CLOCK EXAMPLE')
    1         NUMOUT: DO;
              /****************************************************************
              * THIS PROCEDURE CONVERTS A BINARY VALUE (BYTE OR ADDRESS) TO AN
              * ASCII CHARACTER STRING THAT REPRESENTS THE VALUE OF THE NUMBER
              * IN A SPECIFIED BASE.
              * PARAMETERS:
              *   VALUE:   THE BINARY NUMBER TO BE CONVERTED
              *   BASE:    BASE 'VALUE' IS TO BE REPRESENTED IN (2,8,10,16)
              *   LC:      LEFT FILL CHARACTER TO BE SUPPLIED IF THE LENGTH
              *               OF THE CONVERTED STRING IS LESS THAN THE
              *               RECEIVING FIELD
              *   BUFADR:  STARTING ADDRESS OF THE FIELD IN WHICH THE ASCII
              *               REPRESENTATION IS TO BE PLACED.
              *   WIDTH:   NUMBER OF BYTES IN THE RECEIVING FIELD
              ****************************************************************/
    2   1     NUMOUT: PROCEDURE(VALUE,BASE,LC,BUFADR,WIDTH) PUBLIC;
    3   2        DECLARE (VALUE, BUFADR) ADDRESS;
    4   2        DECLARE (BASE,LC,WIDTH,I) BYTE;
    5   2        DECLARE CHARS BASED BUFADR(1) BYTE;
    6   2        DECLARE DIGITS(*) BYTE DATA ('0123456789ABCDEF');

    7   2        DO I = 1 TO WIDTH;
    8   3           CHARS(WIDTH-I) = DIGITS(VALUE MOD BASE);
    9   3           VALUE = VALUE / BASE;
   10   3           END;
   11   2        I=0;
   12   2        DO WHILE CHARS(I) = '0' AND I < WIDTH - 1;
   13   3           CHARS(I) = LC;
   14   3           I = I + 1;
   15   3           END;
   16   2        END NUMOUT;
   17   1     END;
```


MODULE INFORMATION:

      CODE AREA SIZE   = 00AAH   170D
      VARIABLE AREA SIZE = 0008H   8D
      MAXIMUM STACK SIZE = 0006H   6D
      33 LINES READ
      0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION